DOCUMENT RESUME

ED 257 414 IR 011 599

AUTHOR Burrowes, Sharon; Burrowes, Ted
TITLE Improving CAI in BASIC.
INSTITUTION International Council for Computers in Education, Eugene, Oreg.
REPORT NO ISBN-0-924667-08-7
PUB DATE 85
NOTE 91p.
AVAILABLE FROM International Council for Computers in Education, University of Oregon, 1787 Agate Street, Eugene, OR 97403-1923 (1-4 copies $6.00 each, 5-9 copies $5.40 each).
PUB TYPE Guides - Non-Classroom Use (055) -- Computer Programs (101)

EDRS PRICE MF01 Plus Postage. PC Not Available from EDRS.
DESCRIPTORS Authoring Aids (Programing); *Computer Assisted Instruction; Courseware; *Design Requirements; *Epistemology; *Instructional Design; *Production Techniques; *Programing; Programing Languages; Responses
IDENTIFIERS *BASIC Programing Language

ABSTRACT

This booklet is designed to help users understand the skills and concepts needed to develop effective computer assisted instruction (CAI) programs in any computer language and to polish and extend BASIC language programming skills. It assumes some idea of CAI and some BASIC programming (but not fluency), and can be used for self-study, in a workshop, or as a text in a computer education course. Examples are written in Applesoft BASIC programming language, with footnotes assisting for other BASIC dialects. An introductory chapter establishes the context for the rest of the booklet, and the second chapter gives an overview of the ways in which traditional learning theory interrelates with the writing of CAI. Each of the remaining chapters deals with a factor that should be considered when writing CAI, including controlling the screen, checking answers, user control, providing appropriate tasks--controlling randomization, rewards, and response verification. Appendices include a computer literacy review program, "case" structure for multiple tries, multiple re-asking in response to wrong answers, a music routine, and an approximately timed input interrupt routine. (LMM)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

- [x] This document has been reproduced as received from the person or organization originating it.
- [ ] Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

"PERMISSION TO REPRODUCE THIS MATERIAL IN MICROFICHE ONLY HAS BEEN GRANTED BY

D. Morsund

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

About the Authors

Dr. Sharon Burrowes is currently a computer science teacher and computer resource teacher for the Wooster (Ohio) Public Schools. For the past few years, she has been deeply involved in teacher training both locally and at a number of universities in northeast Ohio. Most recently she has become part of an authoring team developing a series of books on Logo for CBS Educational and Professional Publishing.

Dr. Ted Burrowes has taught mathematics at the junior high school level and computer science and mathematics at the college and university level. Although currently a systems analyst for the Ohio Agricultural Research and Development Center of the Ohio State University, he has long been interested in the uses of computers in education. He currently pursues his interest in educational computing primarily through the development of educational software for Burrowes Software.

This booklet is published by the International Council for Computers in Education, a non-profit, tax-exempt professional organization. ICCE is dedicated to improving educational uses of computers and to helping both students and teachers become more computer literate. ICCE publishes *The Computing Teacher*, a journal for teachers and for teacher educators. It also publishes over 15 booklets of interest to educators.

The booklet prices given below are for **prepaid** orders. On other orders a $2.50 processing charge will be added.

QUANTITY	PRICE (U.S. $)
1-4 copies	$6.00 each
5-9 copies	$5.40 each
10-99 copies	$4.50 each
100+ copies	$4.20 each

Please place your orders with:

INTERNATIONAL COUNCIL FOR
COMPUTERS IN EDUCATION
University of Oregon
1787 Agate St.
Eugene, OR 97403-1923
(503) 686-4414, 686-4429

Copyright © ICCE 1985
ISBN 0-924667-08-7

BEST COPY AVAILABLE

IMPROVING CAI IN BASIC

Preface 3

1. INTRODUCTION 4

2. LEARNING THEORY AND PROGRAM DESIGN 5
2.1 Introduction 5
2.2 Problem Analysis 7
2.2.1 An Introduction 7
2.2.2 Example 1—Adding Fractions 8
2.2.3 Example 2—Computer Literacy Review 9
2.2.4 Pseudocode, Subroutines and Testing 12
2.3 Learning Theory 13
2.3.1 Feedback 13
2.3.2 Individualization 15
2.3.3 Locus of Control 16
2.3.4 Distractors 17
2.4 Programming Style Considerations 17
2.5 Getting Started 18

3. CONTROLLING THE SCREEN 20
3.1 Introduction 20
3.2 Clearing the Screen 22
3.3 Positioning the Cursor 22
3.4 Controlling the Speed of Presentation 25
3.5 Some Special Effects 26

4. CHECKING ANSWERS 28
4.1 Code to Check Answers (IF-THEN-ELSE) 28
4.2 Giving the Correct Answer 31
4.3 Giving a Second Try (WHILE Loop) 33
4.4 End of Section Evaluations 37

5. USER CONTROL 39
5.1 Using Menus 39
5.2 An Aside—Some Problems with Data Statements 42
5.3 Student Control of Page Turning 45
5.4 Number of Questions 47
5.5 Choosing the Speed of Presentation 49
5.6 User Friendliness 50

6. PROVIDING APPROPRIATE TASKS—
CONTROLLING RANDOMIZATION 52
6.1 Randomization at Power Up 52
6.2 Randomization with Numbers 53
6.3 Randomization with Strings 55

7. REWARDS 60
7.1 Considerations in Choosing and Designing Rewards 60
7.2 Using Randomization with Rewards 62
7.3 Using Different Levels of Rewards 63
7.4 Less Obvious Rewards 64

8. RESPONSE VERIFICATION 65
8.1 Examples of Response Checking/Verification 65
8.2 The "REENTER" Problem 67
8.3 Special Cases 69

9. POSTSCRIPT 70

Appendix A: COMPUTER LITERACY REVIEW PROGRAM 71
Appendix B: "CASE" STRUCTURE FOR MULTIPLE TRIES 80
Appendix C: MULTIPLE RE-ASKING IN RESPONSE TO WRONG ANSWERS 82
Appendix D: MUSIC ROUTINE 84
Appendix E: APPROXIMATELY TIMED INPUT INTERRUPT ROUTINE 85

PREFACE

The purpose of this booklet is twofold:

- To help you understand the skills and concepts needed to develop effective computer-assisted instruction (CAI) programs in any computer language.
- To polish and extend your BASIC language programming skills.

This booklet assumes that you have some idea of what constitutes CAI. It also assumes that you have already done some programming in the BASIC programming language, but it does not assume that you are fluent in its use. It is intended to help you expand your knowledge, skills and confidence in both areas.

This booklet can be used for self-study, in a workshop, or as a text in a computer education course. The examples are written in Applesoft BASIC, but footnotes are included to assist readers who are using another BASIC dialect.

The overall organization of the booklet is as follows:

Chapter 1, the introduction, establishes the context in which the rest of the booklet is written and should be read first by every reader.

Chapter 2 gives an overview of the ways in which traditional learning theory interrelates with the writing of CAI. It is important reading for anyone interested in the development of CAI and is referred to in later sections.

The remaining chapters are parallel in structure. Each deals with a factor that should be considered when writing CAI. While it is not necessary to study these sections in the given order, that sequence reflects the order in which questions arise as you start to develop a particular program. Furthermore, program segments assume familiarity with the techniques introduced in previous chapters. So unless you have already written some CAI programs, you will find the booklet flows more smoothly if you study the chapters in order.

CAI 1

INTRODUCTION

Have you examined some CAI materials and thought, "I wonder if I could write programs for my students to use?" Perhaps you feel that the available programs you have tried are too restrictive, not providing the desired features or topics you want. Perhaps you have seen programs that you would like to have, but they were too expensive for your (possibly nonexistent) budget. Perhaps you've been unable to find programs on a topic you want to teach. Or, perhaps you believe that you can write programs at least as good as those presently available. You may want to publish the CAI materials that you are planning to develop.

It is also possible that you have taken a course on BASIC to prepare yourself to write programs. If it was a college or university computer science class, you may have found that the required programs had nothing to do with either education or your particular field of interest. Perhaps you are wondering how all your course material can be used to help write good CAI. Maybe you took a computers-in-education workshop only to find that the workshop was too short to learn what you want and need to know.

In any case, you are probably looking here for practical and useful information. This booklet provides a guide to many of the most important ideas of CAI. It is a practical, brief introduction to the design and implementation of CAI materials.

As you work with computer-assisted instruction, it is important to keep one idea firmly in mind. The goal of CAI is to help students learn more efficiently. There is a large and always increasing body of information about how students learn. Good CAI material combines such knowledge with a thorough knowledge of the subject at hand and with empirical observations. Because the knowledge of how students learn is so important, this booklet begins with a survey of some of the major ideas of learning theory which relate most strongly to CAI.

CAI 2

LEARNING THEORY AND PROGRAM DESIGN

2.1 INTRODUCTION

Somewhere in your educational training you probably had some coursework in behavioral learning theory. You discussed rewards and punishments, shaping and fading, and other such technical ideas. You apply many of these ideas quite naturally in the classroom without ever thinking about them. The same concepts can be applied to decisions about how to most effectively achieve your goals using the computer.

In the classroom you frequently seek ways to reward students. You avoid punishments as best you can and try to provide positive reinforcement whenever possible. When designing CAI, careful thought about reinforcement is very important. Picture a student who is working at a computer. S/he does not have you there to provide feedback or encouragement. Most CAI programs must therefore provide some measure of reinforcement.

In addition, when you work with students in the classroom, you modify your interactions with the class as you see how much difficulty they are having with the topic being studied. What you are doing is using **prompts** *(questions, hand motions, facial expressions)* to **shape** *(modify)* their responses. You gradually **fade** *(remove)* the prompts as they learn. These ideas are essential in any CAI. You will have to anticipate as many student responses as possible and build the teaching part of your program around those responses as carefully as you would in the classroom.

Perhaps in your study of programming or in reading current magazine articles, you have seen references to "top-down design," "step-wise refinement" and "modular testing." These refer to strategies which help make the task of writing good computer programs simpler and make the resulting program easier to read and modify. *Top-down design* refers to looking at the problem and examining what major tasks need to be completed to solve the entire problem before getting into the details of the

solution. *Step-wise refinement* refers to breaking the problem into increasingly small sub-tasks until you reach a manageable size for which to write a section of program code. *Modular testing* means testing these small sections of code as you go along rather than waiting until the entire program is complete. It is as important to apply these ideas when you are writing your first CAI program as when you finally begin writing that really super program that has been in the back of your mind for some time!

As you progress through this booklet, notice that these ideas are woven together. Questions of how to achieve a particular educational objective lead to methods of problem analysis. Language features are related to the classroom activities you seek to simulate. Your experience and ideas find clarification and direction through the formalities of program style and distillations of learning theory.

As new ideas are discussed, we will sometimes use the vehicle of a continuing example for analysis and illustration. This program presents practice on as many as 20 terms/facts that might be taught in a computer literacy unit. It could be used for reinforcement during the unit or as a review or quiz at the end of the unit. A complete listing of the program is found in Appendix A. As a preliminary look at this program, Figures 1 and 2 below show the opening screen and a screen showing a typical question.

```
I ========================================= I
II                                         II
II     COMPUTER LITERACY REVIEW            II
II                                         II
I ========================================= I

THIS PROGRAM CAN BE USED TO REVIEW
ALL 20 TERMS AND FACTS OR TO GIVE A
SHORTER RANDOM QUIZ. IF THIS IS TO
BE A QUIZ, CHOOSE THE NUMBER OF
ITEMS (1-20) TO DO ACCORDING TO YOUR
TEACHER'S INSTRUCTIONS.

NUMBER OF ITEMS?_____
```

Figure 1.

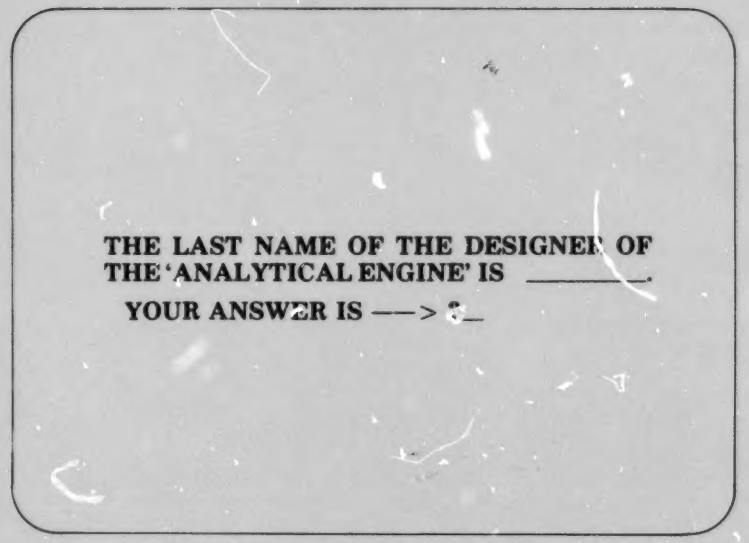


Figure 2.

2.2 PROBLEM ANALYSIS

2.2.1 An Introduction

The first thing to do when you decide to write a CAI program is to be sure you thoroughly understand the program's task. As you well know, any computer programming task works out better if you take the time to plan it carefully in advance. This is as true for good CAI as for any other programming task.

The first step in planning your CAI application is to do some form of what learning theorists call a "task analysis." Sources differ as to what exactly should be included in a task analysis and how detailed it should be. Our ongoing example does not require the degree of detail often associated with the term "task analysis." To avoid confusion, we will simply refer to our efforts as a **problem analysis.** Whatever the project, you should tailor the depth of analysis to the type of problem being solved and to the nature of the students that will be using the materials.

Whatever depth of *problem analysis* you do, it needs to be done with a great deal of care. The importance of this can be underscored by your imagining a student using your materials. Imagine the student's reaction when coming across an incompletely developed concept or finding that the program refuses to move on in the lesson. It is not hard to predict the negative

educational impact of such a frustrating experience. Careful planning is needed to be sure such events do not happen.

A thorough analysis is easier to do if you are writing a program to teach a topic that you have often taught before. But do not let familiarity breed haste, even in your most comfortable areas. Careful planning of the program details cannot be overestimated.

One possible form of such a problem analysis is as follows:

A. Determine the entering behaviors (the prerequisite skills and concepts) by either pretesting or observation.

B. Clearly describe the task(s) involved in completing the instructional unit by:

1. **Breaking the material to be presented into small but coherent pieces;**
2. **Determining which of these pieces must be presented in a particular sequence; and**
3. **Examining the pieces and groups of pieces to find where multiple paths through the overall topic are possible and desirable.**

C. Determine the terminal behaviors (new skills and concepts) that you wish students to have.

This analysis is quite similar to setting up performance objectives for a particular task. However, the focus here is on the detailed activities that a student will perform when using your program.

2.2.2 Example 1—Adding Fractions

The first step in any analysis is to decide what entry-level skills and concepts the program users need. For example, reading level must be considered in any program. Otherwise, it is possible to end up with a good program that is unusable because the students cannot even read the directions.

As an example that we will pursue further, consider a program to teach addition of fractions. The prerequisite skills and concepts here include addition of whole numbers and the fraction concept. You might also include reducing fractions to lowest terms and raising fractions to higher terms, and changing fractions to mixed numbers and mixed numbers to fractions. Some of these might be topics you may need to teach as entry-level skills and concepts.

This leads naturally to the next step, the breaking down of the problem into manageable sections. First, you would probably introduce adding like fractions. Next would be the idea of finding common denominators using what you just reviewed about raising fractions to higher terms. Next would come adding unlike fractions where the answer does not need to be reduced. Then, problems whose answers are fractions that need to be reduced should be given. Finally, problems whose answers need to be changed to mixed numbers should be included.

A comprehensive program also needs to provide problems that include mixed numbers among the original terms. Handling this as a separate subtask would be sensible, and then it would need to be subdivided. First, problems having answers needing no reducing should be given. Then problems with answers that have fractional parts needing reducing would be introduced. The final problems would be those with answers whose fractional parts must be changed to mixed numbers and combined with the whole number part of the answer.

Looking back over this analysis, notice that the border between entry-level skills and some of the fundamental skills which might be taught in the program is fuzzy. This is not uncommon. For this reason, it is often a good idea to consider including some review of prerequisite topics. Such a review could be optional or designed as an integral part of the overall lesson.

As you can see, what initially looked like a fairly simple task turned into one with many parts, each of which would require a lot of thought to turn into a computer program.

The process of breaking down an initial problem into subtasks, which is illustrated by the above example, is part of the *top-down analysis* that we mentioned earlier. This process is also the second step in the three-step problem-analysis outline given in the section 2.2.1 above. Whatever type of programming task you decide to work on—drill and practice, tutorial or simulation—you need to do this kind of initial problem breakdown. Once that is accomplished, the precise way in which you plan to present material must be made very explicit in your thinking.

2.2.3 Example 2—Computer Literacy Review

As indicated earlier, we'll be developing a program to provide practice with a list of computer literacy terms/facts. This program is a rather simple example. It was chosen because it can illustrate many of the concepts to be examined and can also be completed within this booklet. The breakdown into subtasks and possible paths through the program will not get too complex because drill and practice is a relatively simple form of CAI.

Recall that the first step in any problem analysis is to determine entering behaviors. For this program, assume that students have had some experience with the keyboard and have already studied the terms presented by the program. Further, the program will be aimed at junior high students.

Since this is a simple drill-and-practice program, there is only one main section, namely the one that presents questions and checks answers. However, there is also one smaller task that we'll need to consider later: establishing the complete bank of questions and answers.

The overall breakdown of the main section of the problem into subtasks is as follows:

Ask how many questions the student wants to practice.
Establish the questions to be asked.
Repeat, for each question chosen:
  Present the question;
  Accept student's answer; and
  Process student's answer.
Summarize the results.

The next step is to examine each point in this outline in more detail, focusing on the possible alternatives and the reasons for the choices made.

In deciding how many questions to present, there are at least three alternative strategies. One is to prescribe the number of questions and present each one until the student gets it right. A second approach would be to present a certain number of questions and then continue presenting questions until the student achieves above a certain percent correct. In this example the student is expected to enter the number of questions to be asked. This choice was made since this program is one that can be used by several different teachers. Each of them might have a different context in which they would use a quiz of this type.

Once a question is presented, you must decide what is an acceptable form for the answer. Will the answer have to be spelled exactly right or will the program accept it if only the first several letters are correct? In this example only correct spelling is accepted since correct spelling of the computer vocabulary is a program objective.

The example also brings out a frequently encountered issue. Will the student be given a second chance at a question s/he has

missed? If this program were to be used as a final test on a unit, a teacher would not want to allow a second try. If, on the other hand, it is used as a tool for review and reinforcement, allowing multiple tries would make sense on any questions the student didn't get right the first time. In the example two tries are allowed so that a student may make a careless or typing error without penalty, but still receive reinforcement for a correct answer.

Yet another consideration is the reinforcement or reward structure. Rewards can vary from being very simple to very complex. The example program contains a variety of rewards, ranging from single words through text-graphics (pictures or designs made up of keyboard characters) to colorful low-resolution graphics. The reason for having such a variety and the method of selection among them is elaborated in section 2.3.

The outline for the example also includes a summary of results. Will this summary be on the screen or on the printer? Will we want to store the results permanently so that the teacher can look them over later? Will the report include a list of questions missed so the student can study them? For the sake of simplicity and since this program is primarily for review, the final report will contain only the number of questions requested, the number of questions the student got right on the first try, and the number right on the second try. It will also simply display the summary on the screen.

To summarize the more detailed design for the computer literacy review program, here is a more detailed outline.

```
Allow the student to choose the number of questions to be
presented.
Repeat the following for each question selected:
  Present definition of computer literacy term;
  Accept student's answer;
  IF answer is correct—
    THEN give a reward;
    OTHERWISE (i.e. if the answer is wrong)—
      IF it's the first try—
        THEN give a second chance;
        OTHERWISE give the correct answer;
      END for first/second try;
  END for correct/incorrect answer.
END for present and check each question.
Tell student how well s/he did on the set of questions.
```

This form of outline is often called "pseudocode." It is neither English nor a computer language. *Pseudocode* is written in a way that makes it both easy to read and understand and easy to translate into an actual computer language. Section 2.2.4 explores the use of pseudocode further.

As you can see, even this simple drill-and-practice program requires a lot of thought and planning. As you begin your work in CAI, be sure that you pick a relatively small programming task because even the simplest task has many factors to consider.

2.2.4 Pseudocode, Subroutines and Testing

As part of the thorough analysis of the problem, you should always write it in pseudocode as introduced in section 2.2.3. Resist the temptation to begin coding at this point. Work on refining and expanding your pseudocode—it is an excellent way to highlight loops, branches and subsections. It is also an excellent way to examine the educational soundness of your initial ideas. The detail in which you write the pseudocode at any particular time depends on how far you are along in your problem analysis and how complex the original problem is. You will develop confidence in your design as you work with the pseudocode and discover and eliminate the bugs that are bound to be there. Once your pseudocode has stabilized, you'll know that you are ready to begin coding. In section 2.3.1 the pseudocode from section 2.2.3 is expanded as we delve further into the details of that example.

When you are ready to begin coding in BASIC (or any other language), the idea of *modular testing* mentioned earlier comes into the picture. Remember that this technique refers to testing modules (logically cohesive sections of code) as they are written. Subroutines are often a natural vehicle for packaging the code for separate modules, especially for the innermost or lowest-level modules. Higher-level modules may or may not be packaged as subroutines and often call on subroutines containing lower-level modules. Also notice that the use of pseudocode often helps clarify a program's logic in such a way that both high-level and low-level modules are more easily identified.

So don't just code from beginning to end. Instead, begin with the logic which is at the innermost levels. As you finish a module, test it. Then continue this approach, gradually working out to the larger modules of your design. Keep doing that as you develop your program. If problems develop as the program gets more complicated, don't hesitate to go back to small modules and retest them. Your debugging process will go much more smoothly, and your final program will be much more reliable.

Another point to keep in mind is that in a well-structured program there is one entry and one exit point for each module. That

is, you should always enter a subroutine or block of code at the first line of that module, and you should always exit at the last line. Do not write code that jumps into the middle of modules or has several RETURNs in a single subroutine. In the same way, the program itself should have only one END statement.

When writing a subroutine in BASIC, there is one especially convenient way to test the subroutine by itself. Suppose you have a subroutine that begins at line 1000 with a RETURN in line 1999. You can type "Run 1000," and the subroutine will be executed as written. When it reaches the RETURN statement, however, a RETURN WITHOUT GOSUB error message will appear on the screen. This can simply be interpreted as "your subroutine is done," but if it bothers you, you can temporarily replace the RETURN with an END (but then be sure to put the RETURN back before you continue).

2.3 LEARNING THEORY

As noted in the introduction to this booklet, there is a substantial body of information about how people learn. What follows is a more detailed look at some of the major ideas that are applicable to the development of CAI.

Many ideas from behavioral and cognitive psychology are applicable. We could take a detailed look at such concepts as *prompting, shaping* and *fading*. We could also explore the developmental levels through which students pass. We could examine how these levels affect the students' entering behaviors which we wish to assume. What is known about the learning of concepts is certainly relevant to many types of CAI. However, this is not a textbook on learning theories, so we will limit this booklet to the following ideas:

- *Feedback* (rewards/punishment)
- *Individualization*
- *Locus of control*
- *Distractors*

2.3.1 Feedback

Have you ever wished that you had the time to observe each student in your classroom and tell each one how s/he is doing? One of the positive things about using a computer to teach is that students can find out immediately whether or not they are correct. That is to say, the computer can provide immediate feedback. The type and frequency of feedback to be given are important areas for consideration as you plan your programs. Chapter 4 (CHECKING ANSWERS) deals with some aspects of providing feedback.

Another important part of teaching is rewarding students for success. The computer allows you to provide more immediate and more frequent rewards for your students. This reward structure also needs careful planning. Learning theory indicates that immediate and powerful rewards enhance learning. So you need to plan for students to receive powerful rewards. You need to be careful, however, not to punish students for wrong answers with comments that make them feel stupid or foolish.

Novices often choose to begin with a smile face for success and a frown face for failure. A problem can easily arise from such a choice. What if the student prefers frown faces, or what if the frown face is presented in the student's favorite color? It is important that rewards for success be powerful, but responses for failure be as neutral as possible. Usually the best way to deal with wrong answers is a straightforward statement such as, "No, try again" or "The correct answer is. . . ."

Learning theory also indicates that intermittent reinforcement is the most powerful method of reinforcement. Indeed, that is why teachers can manage the classroom—students are willing to keep trying for the occasional smile, pat on the back or good grade. You can build such a reinforcement schedule into a CAI program by planning for a variety of levels of response.

The following is a portion of the pseudocode from section 2.2.3 describing the computer literacy review program. It includes the student's getting a second chance at any missed question.

```
IF answer is correct--
  THEN give a reward,
  OTHERWISE (i.e. if the answer is wrong)--
    IF it's the first try--
      THEN give a second chance;
      OTHERWISE give the correct answer;
    END for first/second try;
END for correct/incorrect answer.
```

We wish to expand this logic further to include more details of the reward structure. Though the reasons for our choices are developed in more detail in Chapter 7 (REWARDS), we use two strategies to gain some of the effect of intermittent reinforcement. One is to use more powerful rewards for answers which are correct on the first try. The second is to use randomization in the selection of rewards. The following pseudocode makes these ideas more specific:

```
IF the answer is correct—
  THEN—
    IF it is the first try—
      THEN—
        Randomly select a number from 1 to 3;
        IF it is 1 THEN give a graphics reward;
        IF it is 2 THEN give first text-graphics reward;
        IF it is 3 THEN give second text-graphics reward;
      OTHERWISE (the student needed the 2nd try)—
        Randomly select a number from 1 to 3;
        IF it is 1 THEN give text-graphics reward;
        IF it is 2 or 3 THEN randomly select a one-word
          reward;
    END for first/second try.
  OTHERWISE (the student missed the answer)—
    IF it is the first try—
      THEN—
        Give the message: "Try again";
        Provide second try;
      OTHERWISE (the student has had a 2nd try)—
        Give a message: "No, the answer is. . . .";
    END for first try;
END for correct answer.
```

Take some time to compare the expanded version with the original. This is an example of one step in a *stepwise refinement* of the original logic outline. Note that it is not yet fully refined since we have not addressed the question of just how to provide the second try nor exactly what the rewards will be. As you perform a stepwise refinement of your own problem analysis, you will probably follow similar patterns, even if your problem is quite different.

2.3.2 Individualization

If you are a classroom teacher, you must have had many frustrating days because you found yourself teaching to the middle of the class. You know some of your students are falling behind and becoming frustrated because they need more explanation and practice than you are giving, while others are bored because they don't need the review that you are providing.

Much of the frustration for both types of students can be alleviated with the use of the computer. Students respond positively to being able to work at their own pace, spending more

time reviewing when needed and skipping material they already know. By including small pretests in CAI material, you can allow brighter students to move ahead more rapidly. By including an opportunity for extra practice, you can also allow the less able students to learn the material at their own rate.

In addition, it is possible to have the computer monitor each student's progress and present material that meets individual needs. This can be done by keeping track of the percent of questions missed and returning the student to a review section when his/her percentage falls below an acceptable level.

Another way to individualize is to introduce variety in tasks presented. For example, a tutorial on adding whole numbers can provide different problems. If the order of presentation of several sections is not important, you can write your program to provide a partially randomized path through them. Such ideas may enhance interest and help prevent boredom.

Our computer literacy example is too limited to explore all of these considerations. Nonetheless, we will include variety in both the selection and order of presentation of the questions.

2.3.3 Locus of Control

Do you remember your first encounter with a computer? Was it a frustrating experience for you? Many adults describe their initial reaction to working with a computer as feeling out of control and helpless. How do you react when you feel that way? Most of us avoid such situations in favor of situations that make us feel secure.

Our students are no different. Haven't you watched students who will do *anything* to avoid a math assignment or a writing exercise? Have you thought about why those particular students react that way? Likely the assignment they are avoiding is one which gives them difficulty. On the other hand, those same students may bubble with excitement over an art project or a science experiment where they feel that they know what they are doing.

The term *locus of control* may be new to you, but is one which is quite important. *Locus of control* refers to a student's feeling that s/he is in control of her/his own learning. A lot of early research on CAI focused on the subject's locus of control. Researchers have found that students feel better about themselves if they are in control of their own learning and that CAI can enhance this improved self-image.

Also notice that lack of locus of control can be seen as part of the anxiety students feel when faced with a hard assignment or that novices feel when they first sit down at a computer keyboard.

The relevance of locus of control for CAI is that students need

to have some control over their learning when working with the computer. To give students some control is one of the reasons that students are asked to choose the number of problems that they want to do in the computer literacy review program.

There are a number of ways to make students feel more in control when using your program. Some of these may be as simple as giving students choices on things such as whether or not there will be graphics or sound rewards. Others can be more subtle, like allowing the student to control the speed at which the material is presented.

Because of the importance of locus of control, Chapter 5 (USER CONTROL) is devoted entirely to a more detailed look at various ways of providing user control in CAI programs.

2.3.4 Distractors

Have you ever tried to write a computer program in a room filled with noise or music? Did you find it difficult to concentrate on the logic of your program? Do your students ever complain about noise in the halls or the humming of the student in the next desk? Learning theorists use the term *distractors* to refer to any factor which *distracts* a learner from the learning task.

When you teach a topic, you minimize the other things going on that might prevent your students from paying attention to the lesson. Similarly, when you design CAI, you need to consider what things might distract students from focusing on what you want them to be doing in your program.

As you write CAI, remember that students will be looking at a screen. Good screen formating avoids visual distractors. Careful thinking about your screen design is very important in the development of good CAI. Chapter 3 (CONTROLLING THE SCREEN) focuses on just such concerns.

The use of sound and graphics in your program can be powerfully positive *factors*. They can also serve as powerful *distractors* if they become an end in themselves and do not serve your educational objectives. It is important that you examine all aspects of your program for possible distractors.

2.4 PROGRAMMING STYLE CONSIDERATIONS

Before beginning to explore specific programming techniques and examples, we digress for a moment into some considerations of program style. These ideas are borrowed from structured programming languages and clearly make BASIC a more readable language. The specific guidelines used in the examples in this booklet are the following:

- **Long variable names will be used.** Even though most versions of BASIC only pay attention to the first two char-

acters, many permit using additional characters. These can help to explain the intended use of the variables and make programs more readable.

- **Frequent remarks (REMs) will be used.** Not only are they helpful during debugging, but they are invaluable when the time comes to modify or enhance the program.
- **Indentation will be used to highlight the body of each loop and the sections of code that belong to a THEN or ELSE part of an IF.** (This idea will be explained more fully in Chapter 4—CHECKING ANSWERS.)
- **Subroutines will be set off by some marker**—usually a row of characters that will stand out in the listing.
- **In general, all GOTOs will refer to REM statements** so that when reading the program, it is clear where the logic is going.
- **GOTOs will refer to a higher line number unless they are part of a WHILE (or REPEAT) structure.** (Again, an example of these structures will be given in Chapter 4—CHECKING ANSWERS.)
- **All program variables will be listed alphabetically at the beginning of the program along with a brief description of their functions.** Alphabetizing the list helps guard against duplicate variable names and variable names which use the same first two characters and thus would be considered the same in most versions of BASIC. It also makes it easy to find the description of a variable about which a question arises.

Not only are these stylistic rules used throughout this booklet, but you are heartily recommended to use them as well. While some of these may seem artificial if you have not used BASIC structured programming before, with experience you will find that they significantly increase the readability and clarity of programs. This in turn translates into shorter development times.

2.5 GETTING STARTED

So where to start? Well, start small. Pick a very small task for your first program—don't try to design an entire curriculum. Pick something you have taught successfully. Pick something that is appropriate for a CAI environment. For example, start with a simple drill-and-practice program on vocabulary or facts. That is exactly why the computer literacy review program is used for our ongoing example. Once you have written a simple program or two, you will be ready to design more complex programs, and you will have built up some sections of program code that you'll probably use and adapt again and again.

The following sections give more specific details on how to implement some of the previously discussed learning theory ideas. They will focus on specific aspects of coding and the related concerns of structured programming. Examples are chosen to illustrate particular concepts and to provide useful building blocks for your future programming.

Learning theory is an essential aspect of developing good CAI materials. For this reason, you would be well advised to occasionally reread the general principles of program design that are discussed here.

CAI 3

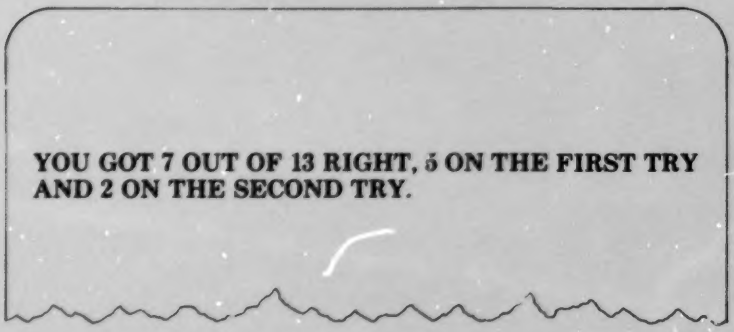
CONTROLLING THE SCREEN

3.1 INTRODUCTION

If you are selecting a textbook, you probably consider the layout of the pages—where information is placed, how easy it is to read, and if it is pleasing to the eye. If you are preparing a student worksheet, you carefully plan not only its content but also its appearance. You should bring these same concerns and skills with you as you plan CAI. Think of the screen as a blank piece of paper to be filled in a neat, readable way.

One important difference between CAI and a textbook is that the screen can be dynamic. For example, you can fill the screen at a controlled rate, and you can cause one piece of information on the screen to change while the rest stays the same. Another difference is cost. That is to say, it isn't necessary to fill up every screen. It's expensive to leave pages of a book partly blank, but the cost of CAI is not changed by partly filled screens. Such capabilities can give you a flexibility in the presentation of material that you have not had before using a computer.

As a simple example, compare the following two screens which might be used to end the computer literacy review/quiz program being developed in this booklet. They contain the same information, but have quite different eye appeal.



YOU GOT 7 OUT OF 13
RIGHT,

5 ON THE FIRST TRY
AND
2 ON THE SECOND TRY

Another factor that can affect the attractiveness of your screen layout is the use of both lower and upper case. Mixed case produces text closer to what students are used to seeing in printed materials. In addition, upper case can then be used for emphasis where, if you could, you would like to use italics or underlining. We have chosen to use upper case for screen text in this booklet, because not all computers have the upper/lower option.

In addition to page layout, there are a number of things to consider when planning several pages of text. It is relatively easy to let students move ahead in the text as they read it, but what if they want to move backward? That will require a different program design. You will have to be careful when you do your problem analysis if you want to be able to move forward and backward through any text you present. This is discussed more fully in section 5.3 (Student Control of Page Turning).

When presenting several pages of text, you need to consider whether or not you want the screen to *scroll*. *Scrolling* means allowing the lines of text to roll off the top of the screen as more come onto the bottom. Would scrolling be distracting? Would it allow students to look back a bit whenever they want? Would it be better to clear the screen between pages and allow students to move forward or backward page by page? Your answers will depend upon your audience and goals.

3.2 CLEARING THE SCREEN

The HOME[1] statement simply clears the screen completely. Every program should begin with a HOME statement so that anything that may have been on the screen from previous use of the computer will be removed.[2] If you decide not to use scrolling, this is the statement you will use to start with a clear screen after each page of text.

3.3 POSITIONING THE CURSOR

The HOME statement gives you a clear screen with the cursor at the upper left-hand corner of the screen. You may not wish to start printing there. The **VTAB** and **HTAB**[3] statements provide for easy movement to other locations on the screen. **VTAB** stands for Vertical **TAB**, and **HTAB** stands for Horizontal **TAB**. That is, the **VTAB** statement directs the cursor to a particular *line* of the screen, and **HTAB** directs the cursor to a particular *column* or *space* across the screen. Thus,

```
100 HOME
110 VTAB 10
120 HTAB 15
130 PRINT "HI"
```

places the word "HI" on the 10th line of the screen and at the

---

[1]Many other microcomputers use CLS or CALL CLEAR to clear the screen. If neither of these work for your machine, check your manuals for screen control commands or clearing the screen. Wherever you see HOME in the following examples, replace it with the statement that your machine uses to clear the screen. If there is none you can use:

```
1000 FOR WHERE=1 TO LINESPERSCREEN
1010    PRINT "<put here as many blank spaces as there are spaces per
        line>"
1020 NEXT WHERE
```

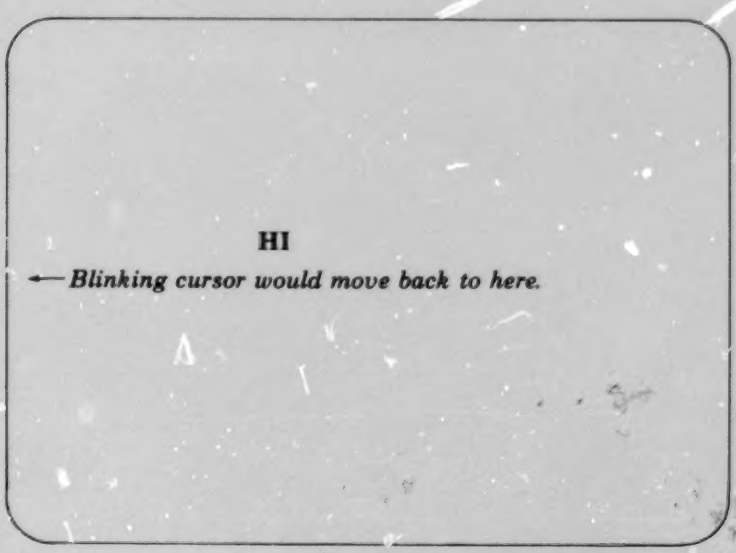
[2]On Apple computers, you may also want to include the TEXT command to clear the graphics screen as well.

[3]Again, most other microcomputers use different statements to place the cursor on the screen. The most common is the PRINT AT statement. However, if your manual doesn't seem to contain such statements, study the following example which should work on any microcomputer:

You can replace VTAB 4 and HTAB 8 with

```
2030 PRINT
2040 PRINT
2050 PRINT
2060 PRINT TAB(8);
```

15th space in from the left. The screen looks something like this after running the above program segment:



These statements can be used to cause selective changes on the screen. For example, suppose you want to print the word "WARNING" in the middle of the screen, perhaps later adding other text above, below and to both sides of it, and then still later want only the word "WARNING" to disappear. You can do that by first writing

```
100 HOME
110 VTAB 12
120 HTAB 18
130 PRINT "WARNING"
```

and then, later in the program when you are ready for the disappearance:

```
1200 VTAB 12
1210 HTAB 18
1220 PRINT "        "
```

Notice that HOME is not appropriate in the second segment because only the word "WARNING" is erased, not the entire screen. Everything else on the screen remains unchanged.

Using a combination of the statements we have just discussed, the beginning of the computer literacy terms/facts program looks like this (and produces the screen shown in section 2.1 above):

```
1000 REM
1010 REM USER INTRODUCTION TO THE PROGRAM:
1020 REM-----------------------------------------------
1030 TEXT
1040 HOME
1050 VTAB 3
1060 PRINT "  I=============================================I"
1070 PRINT "  II                                           II"
1080 PRINT "  II       COMPUTER LITERACY REVIEW            II"
1090 PRINT "  II                                           II"
1100 PRINT "  I=============================================I"
1110 VTAB 10
1120 PRINT "  THIS PROGRAM CAN BE USED TO REVIEW"
1130 VTAB 12
1140 PRINT "  ALL 20 TERMS/FACTS OR TO GIVE"
1150 VTAB 14
1160 PRINT "  A SHORTER RANDOM QUIZ. IF THIS IS"
1170 VTAB 16
1180 PRINT "  TO BE A QUIZ, CHOOSE THE NUMBER OF"
1190 VTAB 18
1200 PRINT "  TERMS (1-20) TO DO ACCORDING TO"
1210 VTAB 20
1220 HTAB 4
1230 PRINT "YOUR TEACHER'S INSTRUCTIONS."
1240 VTAB 23
1250 HTAB 12
1260 PRINT "NUMBER OF TERMS";
1270 INPUT NUMBER$
```

Another way you might want to use HTAB and VTAB is to create action on the screen. Suppose that you want to highlight an idea by having an arrow (— — →) move across the screen from left to right on the 5th line from the top, stopping at column 31. The following code performs that task:

```
110 FOR SPACECTR=1 TO 28
120   VTAB 5
130   HTAB SPACECTR
140   PRINT " -> "
150 NEXT SPACECTR
```

This is an example of a spot in a program that needs documentation. At first glance, it looks as if an arrow would indeed move across the screen, but would leave a trail of dashes. More careful inspection shows you that there is a space before the first "—." That causes the previous "tail" to be erased. It would be most

appropriate to add a remark to this program segment such as:

```
135 REM SPACE BEFORE ARROW TO ERASE PREVIOUS
      ARROW TAIL
```

In addition, if you try this segment of code, you will find that the arrow moves across the screen very rapidly. For a segment such as this, you might want to use a pause routine or the speed option as described in the next section.

3.4 CONTROLLING THE SPEED OF PRESENTATION

One way to control the speed of presentation of text on the screen is to use a *null* (that is, a do-nothing) FOR-NEXT loop. For example, the following segment will cause the word "HELLO" to be printed; then a brief pause before the phrase "TODAY WE ARE GOING TO. . ." appears on the screen.

```
10 PRINT "HELLO"
20 FOR PAUSE=1 TO 500
30 NEXT PAUSE
40 PRINT "TODAY WE ARE GOING TO. . ."
```

Since a program is likely to use several pauses if it uses any, one convenient way to implement a pause loop is as a subroutine.

```
10 PRINT "THE YEAR IS 1632 AD. . ."
20 PAUSE=500
25 GOSUB 10000
30 PRINT "OUR PROTAGONIST IS GALILEO
   GALILEI. . ."
40 PAUSE=2000
45 GOSUB 10000
50 PRINT "HIS PUBLICATION OF A SCIENTIFIC
   TREATISE"

10000 REM **********PAUSE SUBROUTINE
10010 FOR CYCLE=1 TO PAUSE
10020 NEXT CYCLE
10030 RETURN: REM FROM PAUSE
```

This program segment provides a touch of drama by pausing briefly after displaying, "THE YEAR IS 1632 AD. . ." and then pausing about four times as long after "OUR PROTAGONIST IS GALILEO GALILEI. . ." followed then by the subsequent paragraph represented by the line 50.

The use of pause loops allows controlled appearance of text and/or graphics on the screen and can be used with all versions of BASIC. Such uses of pauses make it possible to shape the student's learning behavior. For example, presentations of material can be slowed down if a student has had trouble and is sent back to review a section, or speeded up if the student is

having little or no trouble with new material.

Another way to control the speed at which text is put on the screen is to use the SPEED[4] statement. SPEED=5 causes the text to appear slowly—one letter at a time, while SPEED=150 causes the text to appear at moderate speed. The full range of values that can be used by SPEED is 0 to 255.

The following example uses the SPEED statement to cause the initial greeting to appear letter by letter, but the rest of the message appears at the normal speed that the computer puts information on the screen:

```
10 SPEED=10
20 PRINT "IMPORTANT"
30 SPEED=255
40 PRINT "ALWAYS BE ON TIME"
```

When using the SPEED statement, it is important that you remember to reset the speed to normal (SPEED=255) when you are done with a slower presentation and at the end of a program which consistently uses a slower speed. It can be *very* frustrating for a student to exit a program and find everything happening at a snail's pace.

CAI authors often want to use another kind of pause: one which waits for the student to type a response but will wait for only a certain length of time before going on. A timed-interrupt input loop that will work with Apple II computers is given in Appendix F.

3.5 SOME SPECIAL EFFECTS

When you are working with a piece of paper and want to emphasize something, you usually underline or write in larger letters. There are two easy ways to provide emphasis on computer screens. The **INVERSE**[5] statement causes letters to be printed black on white instead of white on black. The **FLASH**[5] statement causes the letters to alternate quite rapidly between white on black and black on white. The **NORMAL**[5] statement causes subsequent text to appear in the usual white on black form.

To illustrate these two commands, the following program segment causes the screen to display a somewhat dramatic message.

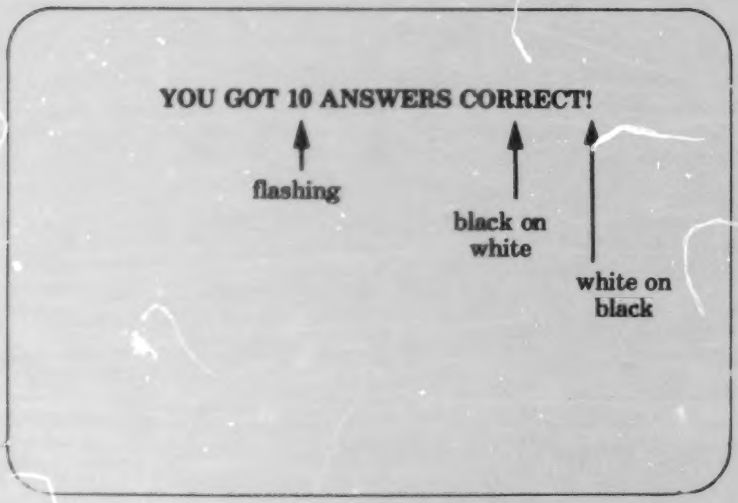
```
100 HOME
110 VTAB 10
120 HTAB 6
```

---

[4]The SPEED statement is not available in many versions of BASIC.

[5]INVERSE, FLASH and NORMAL are not available in some versions of BASIC.

```
130 PRINT "YOU GOT ";
140 FLASH
150 PRINT "10";
160 NORMAL
170 PRINT " ANSWERS ";
180 INVERSE
190 PRINT "CORRECT";
200 NORMAL
210 PRINT "!"
```



It is important that you remember to use the NORMAL statement whenever you are finished with inverse or flashing characters. Otherwise you may end up with an entire screen in inverse and/or flashing characters.

Careful use of screen formatting and INVERSE and FLASH, together with SPEED and/or pause factors can greatly enhance what might otherwise be a rather ordinary and uninspiring text. On the other hand, overuse of such techniques can distract from the point that you are trying to make.

Also consider the possibility of satiation. A learner responds less and less to a particular stimulus as it is repeated. Overuse of the INVERSE and FLASH commands can easily result in satiation. Learners may even cease to notice your use of these for emphasis if you do not use them judiciously.

All of the commands discussed in this section can enhance your presentation of text material if used with care. You should plan each screen of text carefully. Good text presentation can make the difference between a program that looks quite amateur and one that looks very professional.

CAI 4

CHECKING ANSWERS

4.1 CODE TO CHECK ANSWERS (IF-THEN-ELSE)

Once you have decided upon the questions to be asked in your program, you must consider the program's responses to what students give as answers. It is relatively easy to use a statement such as

```
IF STUDENTANS=CRRECTANSWER THEN PRINT "GOOD!"
```

to provide a simple response for a correct answer. You are likely, however, to be dissatisfied with the results because you will probably want your "GOOD" centered on the screen, perhaps a few lines below the student's answer. Two solutions are likely to occur to most people.

First, if your version of BASIC permits it, you might try multiple statements on a line. The result might look like this:

```
IF STUDENTANS=CRRECTANSWER THEN PRINT:PRINT
:HTAB 18:PRINT "GOOD!"
```

While this is adequate for simple circumstances, before very long you will exceed BASIC's limit for the number of characters allowed in one statement. Furthermore, using multiple statements on a line too easily leads to code that is quite hard to read.

The second approach, which many BASIC programmers would use, resembles the following abbreviated program segment.

```
100 IF STUDENTANS=CRRECTANSWER THEN GOTO 900
110          .
             .                    [other program code here]
             .
890 GOTO 950:REM TO SKIP OVER CODE FOR RIGHT
    ANSWER
900 PRINT
910 PRINT
920 HTAB 18
930 PRINT "GOOD!"
940 GOTO 110:REM TO GET BACK TO WHERE WE CAME
    FROM
950          .
             .                    [program continues...]
             .
```

While this solution works, you may be able to see that using this approach can quickly get you into some confusing coding. Before you know it, your program is full of GOTOs that patch things together in bits and pieces. Code like this is often called *spaghetti code* because it becomes so tangled. Spaghetti code becomes increasingly difficult to debug as your program grows longer.[1]

The difficulties we see here are in large measure due to the fact that many versions of BASIC (including Applesoft BASIC) are not structured languages. That is, these versions of BASIC do not offer effective structures to handle problems such as the one above. To help deal with this problem, we can imitate good programming structures that are not available directly. Once you learn to use such structures, your programs will be easier to understand, debug and enhance.

The first of these structures is the so-called **IF-THEN-ELSE**. (Another structure, the **WHILE** structure will be examined in section 4.3 of this chapter.) This structure allows us to say, "If thus-and-so is true, then do these things; otherwise do those things." This is a fairly natural way of thinking, even though we don't phrase it that way in our day-to-day thinking. We think, "If it is raining, I'll take an umbrella, otherwise I'll take a sun hat." In the case of the program we are writing, we are trying to accomplish, "If the answer is correct, then reward the student; else give the student the correct answer." An example of such a program segment is shown below; look especially at lines 300-420.

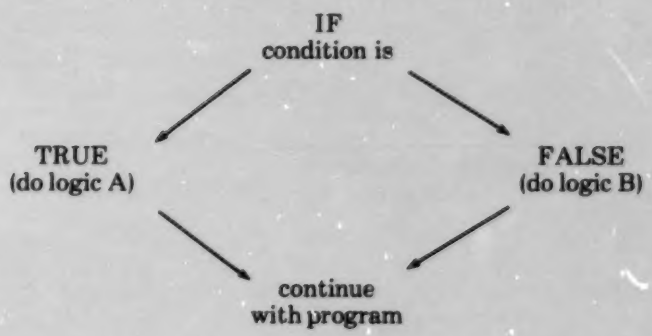
```
100 REM SAMPLE SHOWING AN IF-THEN-ELSE
    STRUCTURE.
110 REM    HERE IT IS USED TO CHECK ANSWERS
120 REM
130 REM VARIABLE LIST:
140 REM    ANS$ -- CORRECT ANSWER
150 REM    QUES$ -- QUESTION FOR STUDENT
160 REM    SANS$ -- STUDENT'S ANSWER
170 REM
200 REM BEGIN MAIN PROGRAM
210 VTAB 10
220 HTAB 5
230 PRINT "WHAT DOES THE TERM 'RAM' MEAN WHEN
    DISCUSSING COMPUTERS"
240 ANS$="RANDOM ACCESS MEMORY"
```

[1]For more on this topic, see "Slicing Through Spaghetti Code" in *The Computing Teacher* for April, 1983 and "Structured Programming in BASIC," a series of articles in the May, June, July and August 1984 issues of *Creative Computing*, all by Arthur Luehrmann.

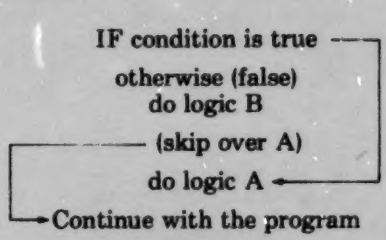
```
250 VTAB 12
260 HTAB 15
270 INPUT SANS$
300 REM ************************************************
310 REM  IF-THEN-ELSE TO CHECK FOR CORRECT
    ANSWER
320 IF ANS$=SANS$ THEN GOTO 380
330 REM  FALSE CONDITION ('ELSE' PART)
340   VTAB 14
350   HTAB 15
360   PRINT "NO, THE CORRECT ANSWER IS ";ANS$
370   GOTO 420
380 REM  TRUE CONDITION ('THEN' PART)
390   VTAB 14
400   VTAB 5
410   PRINT "GOOD!"
420 REM  END IF TO CHECK FOR CORRECT ANSWER
```

Many people, including those who know BASIC but have used only the IF-THEN, find the IF-THEN-ELSE structure a bit confusing when they first encounter it. The major stumbling block for people who see this structure for the first time is the GOTO after the IF . . . THEN. This statement directs the program to the more remote block of code which is the THEN block. In most diagrams and most other computer languages, the block of code for the THEN part immediately follows the IF. The following diagrams may help you see the way BASIC most naturally accommodates the IF-THEN-ELSE.

Conceptually:



Can also be visualized as:



And finally:

```
100 IF condition THEN GOTO 200
110      REM (FALSE)
             •
             •                          [code for logic B]
             •
160      GOTO 300
200    REM (TRUE)
             •
             •                          [code for logic A]
             •
300 REM (END OF IF)
```

Another comment is in order about the programming style used in this example. We are adhering to the stylistic rules we gave in section 2.4 (Programming Style Considerations). Among other things, it calls for the use of many REM statements. If you have never programmed using such extensive REM statements, the above style probably seems unnecessarily cumbersome. For this short segment, that is probably true. But if you have done much programming, you know that programs have a tendency to grow and grow and grow. Before you know it, understanding how your program works can become an impossible task. Proponents of structured programming know that good structured programs may take longer to write initially but in the long run are much easier and faster to debug, maintain and modify. Don't cut corners as you are beginning a project, only to find you have a mess a few weeks later.

4.2 GIVING THE CORRECT ANSWER

At some point in your program, if a student misses a problem, you want to provide the correct answer. Sometimes you want to provide only the correct answer, and sometimes you want to provide a tutorial response. In any case, you should plan your response to incorrect answers at least as carefully as your response to correct answers. Students can become extremely

frustrated working with a computer if they are making errors and do not get help which would allow them to be more successful as they go on. Think of your own non-computer experience. To be able to check an answer key or review a sample problem is very helpful.

Many teachers are tempted to provide witty responses to a student's errors. Think very carefully before you decide to do this. Would you make such a remark to every student in your class, or would you sense that some students might be offended or might misinterpret your humor? Or, might students become so fascinated with the remarks that come with wrong answers that they would give wrong answers on purpose just to see what would happen next? The point here again comes from learning theory. Students respond to positive reinforcement, so that the most powerful responses should be given for correct answers, not incorrect answers. It is extremely easy to make the response to incorrect answers more interesting than you intended, so beware!

Another thing to consider when a student gives a wrong answer is what to do with the wrong answer. Once the student enters an answer, it is sitting there on the screen. Is that what you want? It is possible that you are reinforcing the wrong answer. After all, you probably formatted your screen to be free from distractors, to encourage focusing on just what you want and no more.

With young children you might want to erase the incorrect answer so that they see only the correct answer. A set of sample screens is shown below in which this is done. With older students you might want to direct them to study their incorrect response. In this case, it may help to erase the original response and reprint it elsewhere on the screen. (Section 3.3—Positioning the Cursor—shows a method you might use to blank out an answer.)

Screen as it might look when a question is presented:

HOW MANY STATES ARE THERE NOW IN THE UNITED STATES?

YOUR ANSWER IS:

Screen after student has typed in his/her answer and before s/he has pressed the RETURN key:

```
HOW MANY STATES ARE THERE
NOW IN THE UNITED STATES?

YOUR ANSWER IS: 13
```

Screen as the program responds to the incorrect answer:

```
HOW MANY STATES ARE THERE
NOW IN THE UNITED STATES?

NO, THERE ARE 50 STATES.
```

4.3 GIVING A SECOND TRY (WHILE LOOP)

When you are working on a one-to-one basis with a student who misses a problem, don't you often say, "Try that again?" If the computer is helping a student learn, then the opportunity to try again is often very appropriate. Not only does it provide the opportunity for students to learn more about why they are making mistakes, but it also makes the computer seem more forgiving if the student accidentally presses a wrong key.

Below are two examples of second-try structures. These examples employ what is usually called a "WHILE loop." This is another structure not available directly in many versions of BASIC. The **WHILE loop** is used when you want to repeat an action, but you don't know ahead of time exactly how many times it should be done. You might think, "As long as I'm still hungry, I'll eat another cookie." Similarly, in this instance, you think, "While the answer is not correct, provide a response." Then, when the answer is correct, you will exit the loop and provide an appropriate reward.

The first example uses a string variable to keep track of how many tries a student has had. The second example is extracted from the computer literacy review program in Appendix A. It uses a numeric counter to keep track of the number of tries. To examine the underlying logic involved in both, you should first look at the pseudocode and then the corresponding BASIC code segments.

Pseudocode:

```
Present a question and prompt for an answer;
Set an indicator to keep track of the number of tries;
Accept the student's answer;

WHILE the student's answer is wrong
  Check if this is the first try for this question;
  IF it is the first try, THEN
    Give the message "No, try again.";
    Clear off the incorrect answer;
    Prompt the student to type in a new answer;
    Set the indicator to one wrong try;
  ELSE (that is, if it is the second try)
    Give the message "No, the correct answer is ——";
    Set the indicator to two wrong tries;
  ENDIF;
End WHILE loop to allow second try;

IF the student's answer is correct on the first try THEN
  Give a reward;
ENDIF;
Go on to the next question (or exit if all questions have
been presented.
```

Resulting BASIC code:

```
100  REM EXAMPLE FOCUSING ON A STRUCTURE THAT
110  REM WILL ALLOW A STUDENT A SECOND TRY ON A
         QUESTION
120  REM
130  REM VARIABLE LIST:
140  REM    CANS$ — CORRECT ANSWER
150  REM    FLAG$ — TO IDENTIFY THE ATTEMPT AT
              PROBLEM
160  REM    SANS$ — STUDENT'S ANSWER
170  REM
                 .          [Present question and prompt
                 .                        for answer]
                 .
1190 REM INITIALIZE FLAG$
1200 FLAG$="FIRST"
1210 INPUT SANS$
1220 REM
1230 REM•••••••••••••• WHILE TO CHECK ANSWER
1240 IF SANS$=CANS$ OR FLAG$="BAD" THEN GOTO
     1410
1250   REM
1260   REM IF-THEN-ELSE TO CHECK WHICH TRY
```

```
1270   IF FLAG$ < > "FIRST" THEN GOTO 1340
1280   REM THEN PART – ALLOW SECOND TRY
1290     REM INSERT SUGGESTIONS, ETC. HERE
1300     PRINT "NO, TRY AGAIN!"
1310     INPUT SANS$
1320     FLAG$="SECOND"
1330     GOTO 1380
1340   REM ELSE PART – SECOND TRY WRONG
1350     REM CORRECT ANSWER GIVEN
1360     PRINT "NO, THE CORRECT ANSWER IS . . ."
1370     FLAG$="BAD"
1380   REM END IF TO CHECK WHICH TRY
1390   REM
1400   GOTO 1230
1410 REM END WHILE TO CHECK ANSWER
1420 REM
1430 REM TEST TO PROVIDE REWARD
1440   IF FLAG$< >"BAD" THEN PRINT "GOOD FOR
       YOU!"
1450 REM END TEST FOR REWARD
```

The second example appears very much the same as the first, except that a numeric counter (TRYCTR) is used to keep track of which try a student is on.

```
100   REM EXTRACT FROM APPENDIX A TO ILLUSTRATE
110   REM    'SECOND TRY' LOGIC (WITH AN EMPHASIS
120   REM    ON SCREEN CONTROL)
140   REM VARIABLE LIST
180   REM BLANK$—STRING CONSTANT FOR BLANK
          LINE
210   REM LINECTR—LOOP INDEX FOR PARTIAL
          SCREEN CLEARING
290   REM QUESCTR—COUNTS QUESTIONS PRESENTED
310   REM SANS$—ANSWER INPUT BY STUDENT
330   REM TRYCTR—COUNTS ATTEMPTS AT A PROBLEM
340   REM UANS$—ARRAY OF ANSWERS TO BE USED
360   REM UQUES$—ARRAY OF QUESTIONS TO BE USED
                .                    [Present a question and prompt
                .                                  for answer.]
                .
2350 INPUT SANS$
2360 TRYCTR=1
2370 REM **************************************************
2380 REM 'WHILE LOOP' TO CHECK ANSWER AND
          RESPOND:
2390 IF SANS$=UANS$(QUESCTR) OR TRYCTR > 2 THEN
     GOTO 2750
2400   REM –'IF-THEN-ELSE' TO CHECK WHICH TRY:
2410   IF TRYCTR< >1 THEN GOTO 2580
2420   REM –FIRST TRY WAS WRONG:
```

```
2430      VTAB 14
2440      HTAB 14
2450      PRINT "NO, TRY AGAIN!"
2460      REM  -RETURN-TO-CONTINUE:
2470      GOSUB 40000
2480      VTAB 11
2490      FOR LINECTR=1 TO 13
2500        PRINT BLANK$
2510      NEXT LINECTR
2520      VTAB 11
2530      HTAB 5
2540      PRINT "YOUR ANSWER IS --->";
2550      INPUT SANS$
2560      TRYCTR=2
2570      GOTO 2690
2580    REM  -SECOND TRY WAS WRONG:
2590      VTAB 15
2600      HTAB 8
2610      PRINT "NO, THE CORRECT ANSWER IS:"
2620      PRINT
2650      PRINT UANS$(QUESCTR)
2660      REM  RETURN-TO-CONTINUE:
2670      GOSUB 40000
2680      TRYCTR=3
2690    REM  -END OF IF-THEN-ELSE TO CHECK WHICH
              TRY
2700    GOTO 2370
2750  REM  -END OF WHILE LOOP FOR ANSWER
            PROCESSING
2760  REM  -PROVIDE REWARD IF QUESTION WAS
            ANSWERED CORRECTLY:
2770  IF TRYCTR < 3 THEN GOSUB 10000
```

This second example is longer since it attends to screen control and has the array structure needed by the example. In Appendix A you can also see how this WHILE loop is imbedded in a FOR-NEXT loop to handle the whole set of questions.

One more thing should be said about using WHILE loops in BASIC. The tests for determining that the loop is done should be contained at the beginning, and exit from the loop should occur only from that point. One tenet of structured programming says that each block of code is to have only one entry point and only one exit point. The reason for this tenet is to help avoid spaghetti code.

More can be done in addition to giving a second try. If you use the second-try idea when checking answers, you might want to consider using the current student attempt to tailor your responses. For example, the first time students miss the problem, they might be given a word of encouragement or some suitable hint before being asked to try again.

Here is an example of a computer screen showing a second try in which a hint is given, and the first answer is left for the student's re-analysis.

```
HOW MUCH IS 70% OF 120 ?
YOUR ANSWER IS: 190

YOUR ANSWER SHOULD BE LESS THAN 120—
TRY THE PROBLEM AGAIN.

YOUR ANSWER NOW IS ——
```

When designing your answer-checking logic, your first thought might be to require a correct answer before going on. While there may be a sound pedagogical reason for not progressing in a classroom until a correct answer is given, it can be a disaster in a computer program. What if the student is putting in a blank that is not in the correct answer or simply cannot figure out the correct answer? S/he could never get to the rest of the lesson. You rarely want to put students using your program in such a frustrating situation.

There might be instances where you would want to take the above ideas a step further. For example, you might want to provide the opportunity to try the problem four or five times. In a tutorial program, you might want to allow the student a number of tries, each time providing different hints and/or directions. Such designs require what is called a CASE structure. If you are interested in such a design, see Appendix B for an example.

4.4 END OF SECTION EVALUATIONS

Most CAI programs should include some kind of summary of progress or success at the end of each logical unit and/or after coherent groups of units. Reports such as this fall into two categories: reports for students and reports for teachers.

A report for students at the end of a unit should give them some idea as to how well they have done to date. It could be as simple as giving the number correct or as involved as listing all the questions missed along with suggestions for further study. The report could appear on the screen or on the printer. In addition, the results shown in the report could be used to offer or require either review or continuation to the next topic.

A report for a teacher might be quite different. A teacher might be interested in the sections students completed and their percent right. The teacher might be interested in how each stu-

dent is doing in relation to the class. Here, too, reports can be given in different forms. The teacher might just be interested in a printout of each student's report as verification that the student has completed the assignment. On the other hand, a more complex report might be stored on disk for access at a later time.

It is clear that teachers often have a need for summary evaluations at the end of sections. It is too often overlooked that the more important need is for students themselves. Not only does such a report provide an element of reinforcement, it can also add to student feelings of locus of control as discussed earlier. Furthermore, it can help to reinforce the students' awareness that their education is for them, not for the teacher.

The exact details of what to include in such summaries should be decided on the basis of the unit's objectives, the program's intended audience, and your view of the teachers' needs for information. It is important that you keep in mind what will be most helpful to students as you design reports for them. You certainly don't want to have the reading level so high that the student can't understand the report or to give the results in percentages to the student who has not yet studied percents.

While checking responses may have turned out to be more complex than you expected, experience will demonstrate that using structures such as those described above is not difficult. Keep in mind that your long-term goal is to write programs that are educationally effective, easy to read, easy to debug, and easy to modify and enhance in the future.

CAI 5

USER CONTROL

When you are working with students individually, you are able to respond to the uniqueness of each student. You give each student time to think about a question or read some instructions. You allow each one a chance to work through a problem at his or her own pace. You would never expect all students to react in exactly the same way or at the same speed. Thus, when you are planning CAI, it is often helpful to build into your programs opportunity for students to participate in controlling their own learning. Do you remember the discussion in the introduction about the importance of locus of control? Students simply feel better about themselves and the task at hand when they have some control over that task.

5.1 USING MENUS

As you progress in your writing of CAI, you will find that your programs get increasingly complex and have more and more features. Further, you will find yourself writing related programs that you'd like to combine into a single program.

One way to manage the resulting complexity is to create what is usually called a menu. A menu allows a student to choose what part of the program s/he wishes to use. Below is a sample program segment containing a menu.

```
100 REM MAIN PROGRAM -- MENU SECTION
110 HOME
120 VTAB 5
130 PRINT "TYPE THE NUMBER OF YOUR CHOICE
    AMONG:"
140 VTAB 8
150 PRINT "  1. REVIEW COMPUTER LITERACY"
155 PRINT "           VOCABULARY"
160 PRINT
170 PRINT "  2. REVIEW BASIC LANGUAGE
           STATEMENTS"
180 PRINT
190 PRINT "  3. REVIEW LOGO LANGUAGE
```

```
                    STATEMENTS"
200  PRINT
210  PRINT "  4. I AM FINISHED NOW"
220  VTAB 20
230  HTAB 18
240  INPUT CHOICE$
250  IF CHOICE$="1" THEN GOSUB 1000
260  IF CHOICE$="2" THEN GOSUB 2000
270  IF CHOICE$="3" THEN GOSUB 3000
280  IF CHOICE$="4" THEN GOTO 400
290  IF CHOICE$="1" OR CHOICE$="2" OR CHOICE$="3"
     THEN GOTO 100
300  REM  RESPOND TO AN INCORRECT CODE FROM
     USER:
310  PRINT
320  PRINT "TYPE ONLY A WHOLE NUMBER BETWEEN
     1 AND 4"
330  REM BRIEF PAUSE:
340  FOR PAUSE=1 TO 400
350  NEXT PAUSE
360  GOTO 100
400  REM  USER HAS CHOSEN TO STOP
410  END
1000 REM  ************************************************
1010 REM  CODE FOR COMPUTER LITERACY
1990 RETURN : REM FROM COMPUTER LITERACY
1999 REM
2000 REM ************************************************
2010 REM  CODE FOR BASIC LANGUAGE
2990 RETURN : REM FROM BASIC
2999 REM
3000 REM ************************************************
3010 REM  CODE FOR LOGO LANGUAGE
3990 RETURN : REM FROM LOGO
```

The screen that results from this sample menu looks like this:

TYPE THE NUMBER OF YOUR CHOICE AMONG:

1. REVIEW COMPUTER LITERACY VOCABULARY

2. REVIEW BASIC LANGUAGE STATEMENTS

3. REVIEW LOGO LANGUAGE STATEMENTS

4. I AM FINISHED NOW

Notice that in this example the menu is re-presented after each topic. Putting menus in loops like this is common but is not mandatory. Menus can also be effectively used at more than one level. For example, in a more complex program, you might have submenus for some of the choices on the main menu.

Using menus in your program design accomplishes several important goals. First of all, a menu keeps the screen from getting cluttered with too much explanation. The student can easily see what is available. Secondly, a menu gives the student some control over the learning situation. The student can decide which sections of the program are relevant to his/her needs. For example, the instructions for using the program can be one choice in a menu, and the student can view them only when s/he feels a need for them. Lastly, use of a menu keeps the program design cleaner and easier to follow and debug. This can be important to a busy teacher who wants to improve sections of a program written at an earlier time.

A simple menu requiring numeric choices may also have some positive effects beyond those mentioned above. By asking that the user type in numbers, those who do not know the keyboard will not have to "hunt and peck" so much. In addition, typing one number is easier than typing a word or series of words to select a segment for use. (See also Chapter 8 on Response Verification.)

Keep in mind that a good program protects the user from getting meaningless responses. It is frustrating for a student to be unable to continue because the message being shown on the screen makes no sense or because the question that was originally asked has scrolled off the screen. This means you should test your program for any possible input that students might give. It also means that menus must be clear, stay on the screen until a valid choice is typed in, and give clear responses to any invalid input.

Using menus leads most people towards program modularization. As noted earlier, keeping programs separated into modules is good programming style. With this in mind, as you are designing a program, it is a good idea to use line numbers in some preplanned way. That is, plan to start each new subroutine with 1000, 2000, 3000, etc. Perhaps plan so that each type of routine is located in certain number ranges. For example, you might put all routines using computation in line numbers ranging from 5000 to 10000, all graphics routines between lines 20000 and 30000, and all utility routines after 50000.

One final word about increasing program size is in order here. It can be tedious, time consuming and lead to many frustrating errors if you have to retype a program. This can happen if you want to combine two or more programs or need to reestablish line numbers to provide room in a segment that has run out of line

numbers. For most computers there are programs available which will allow you to renumber all or parts of a program and to merge two different programs to form a new larger whole. Such a program is worth the investment (if it didn't come free with the computer) for anyone who is going to do any substantial programming.

5.2 AN ASIDE—SOME PROBLEMS WITH DATA STATEMENTS

DATA statements are quite easy for most BASIC programmers to use. It is not uncommon, however, for problems to arise when your program uses both menus and DATA statements. Keep in mind that BASIC treats all DATA statements as forming a single list, regardless of where in the program the statements are located. In your modularized program it is deceptively easy to forget this and think as though the DATA statements in one module are somehow separate from others in other modules.

The following example of a program outline illustrates the problem. If this program were to be run as shown, inappropriate questions and answers could occur.

```
100  REM EXAMPLE TO SHOW PROBLEMS WITH DATA
     STATEMENTS
110  REM
120  REM MENU
130  HOME
135  PRINT "KEY:"
140  PRINT "1. FOR FIRST SECTION"
150  PRINT "2. FOR SECOND SECTION"
160  PRINT "3. TO STOP"
170  INPUT CHOICE
180  IF CHOICE < 1 OR CHOICE > 3 THEN GOTO 250
190  ON CHOICE GOSUB 1000,2000
200  IF CHOICE < 3 THEN GOTO 130
210  END
250  REM RUDIMENTARY RESPONSE VALIDATION:
260  PRINT "INVALID RESPONSE; REKEY IT PLEASE"
270  GOTO 140
999  REM
1000 REM ********** FIRST SECTION
          .                    [subroutine for first section,
          .                    including appropriate READ
          .                    and DATA statements]
1990 REM RETURN : REM FROM FIRST SECTION
1999 REM
2000 REM ********** SECOND SECTION
          .                    [subroutine for second section,
          .                    including appropriate READ
          .                    and DATA statements]
```

```
2990 RETURN : REM FROM SECOND SECTION
```

Everything works fine if the program is directed to the first section and then to the second section and then to stop. Any other order causes a problem. If, for example, you ran the program and began by asking for the second section, the questions would be read beginning with the DATA statements in the first section, perhaps never getting to any of the questions for the second section at all. If you did ask for section 1, then section 2, but then went on to request either section a second time, a different problem would occur. More READing would be called for, but there would be no more DATA statements to be read.

There are several possible remedies. The least complicated is to use the RESTORE statement and dummy READs as shown below:

```
100  REM EXAMPLE TO SHOW ONE REMEDY TO READ-
     DATA PROBLEMS
          .
          .                          [lines 110-210 as before]
          .
1000 REM ********************* FIRST SECTION
1010 RESTORE
          .
          .              [subroutine for first section just as before]
          .
1990 REM RETURN : REM FROM FIRST SECTION
1999 REM
2000 REM ********************* SECOND SECTION
2005 REM N1 IS THEN NUMBER OF DATA PAIRS FOR
     SECTION 1
2010 RESTORE
2020 FOR DUMMYCOUNTER=1 TO N1
2030   READ DQUES$,DANS$
2040 NEXT DUMMYCOUNTER
          .
          .              [subroutine for second section just as before]
          .
2990 RETURN : REM FROM SECOND SECTION
```

The first modification is the addition of RESTORE to the beginning of each subroutine. This causes the next READ statement encountered to begin at the first DATA statement in the program. With this, no matter how many times or in what order the student chooses the sections, the first one will be using the correct data.

While the RESTORE solves the problem for the first section, it does not completely solve the problem for the second section.

An additional modification is needed; we have to read past the first section's data (using dummy variables). This positions the READ pointer (the place in the overall DATA list where the next READ will be done) at the beginning of the data for section 2.

This problem is a good argument for putting all the DATA statements needed in a particular program in one place. That way, you know where all the data is and can easily find out what is happening if bugs develop. Below is a further revision of the example with which we have been working, including the RESTORE statement, the dummy READ statements, and better placement of the DATA statements. The program is now structured as follows:

lines 100-120: introduction and all DATA statements for the program
lines 130-999: presentation of the menu
lines 1000-1999: first section beginning with a RESTORE statement
lines 2000-2999: second section, beginning with a RESTORE and dummy read statements

```
100  REM PREVIOUS EXAMPLE REVISED TO SHOW A
105  REM BETTER WAY TO PLACE DATA STATEMENTS
110  REM
115  REM DATA STATEMENTS FOR FIRST SECTION
          .
          .          [all DATA statements for first section here]
          .
120  REM DATA STATEMENTS FOR SECOND SECTION
          .
          .          [all DATA statements for second section here]
          .
125  REM MENU
130  HOME
140  PRINT "1. FIRST SECTION"
150  PRINT "2. SECOND SECTION"
160  PRINT "3. STOP"
170  INPUT CHOICE
180  IF CHOICE < 1 OR CHOICE > 3 THEN GOTO 250
190  ON CHOICE GOSUB 1000,2000
200  IF CHOICE < 3 THEN GOTO 130
210  END
250  REM RUDIMENTARY RESPONSE VALIDATION:
260  PRINT "INVALID RESPONSE; REKEY IT PLEASE"
270  GOTO 140
999  REM
1000 REM ********** FIRST SECTION
```

```
1010 RESTORE
          •          [subroutine for first section, including
          •          the READ statements but not the
          •          DATA statements]

1990 REM RETURN : REM FROM FIRST SECTION
1999 REM
2000 REM ********** SECOND SECTION
2005 REM N1 IS THE NUMBER OF DATA PAIRS FOR SEC-
     TION 1
2010 RESTORE
2020 FOR DUMMYCOUNTER=1 TO N1
2030   READ DQUES$,DANS$
2040 NEXT DUMMYCOUNTER

          •          [subroutine for second section, includ-
          •          ing the READ statements but not the
          •          DATA statements]

2990 RETURN : REM FROM SECOND SECTION
```

Another way to solve the DATA problem would be to read all the data for both alternatives into the appropriate arrays as the first task in the main program. In this case the subroutines would do no reading at all. However, this approach may not be practical if you have a lot of questions and limited memory. The reason is that you would be storing all of the data twice; once in the DATA statements and again in the question and answer arrays. The extra storage may be more than the computer's limited memory can handle.

Most other solutions involve the use of external data files, that is, files that are stored on disks or tapes. Although files are not difficult to use, a study of them goes beyond the scope of this booklet. Files allow essentially unlimited amounts of data to be stored and accessed whenever they are needed. If you write much CAI involving the use of large quantities of data, you will probably want to explore using files.

5.3 STUDENT CONTROL OF PAGE TURNING

One way to allow students to be in control in a computer program is to allow them to do the page turning. That is, when a screen of information is presented, allow the student as much time to read and/or study it as necessary. This is nicely accomplished with a routine such as the one given below in which the student must press the RETURN key to continue the CAI presentation.

```
1000 REM ********** RETURN-TO-CONTINUE ROUTINE
1010 REM MOVE CURSOR TO THE BOTTOM OF THE
     SCREEN:
```

```
1020 VTAB 24
1025 REM SEMICOLON ON THE FOLLOWING LINE
1026 REM KEEPS THE SCREEN FROM SCROLLING UP
1030 PRINT " (PRESS THE <RETURN> KEY TO
     CONTINUE)";
1040 INPUT NNYTHING$
1050 REM CLEAR GRAPHICS SCREEN
1060 TEXT
1070 REM CLEAR TEXT SCREEN
1080 HOME
1090 RETURN : REM FROM RETURN-TO-CONTINUE
     ROUTINE
```

Now, inserting GOSUB 1000 into the program each time a block of information has been presented will put the student in control of the page turning.

You may feel that there is a problem with this because students can then only move forward in the text. If you want the students to be able to page backward as well, then you might alter the code like this:

```
100  REM BEGINNING OF PAGE 1 OF TEXT
          .
          .
          .
200  REM END OF PAGE 1 OF TEXT
210  GOSUB 1000 : REM RETURN-TO-CONTINUE ROUTINE
230  REM BEGINNING OF PAGE 2 OF TEXT
          .
          .
          .
350  REM END OF PAGE 2 OF TEXT
360  GOSUB 2000 : REM BIDIRECTIONAL-CONTINUE
     ROUTINE
370  IF DIRECT$="B" THEN GOTO 100
380  REM BEGINNING OF PAGE 3 OF TEXT
          .
          .
          .
420  REM END OF PAGE 3 OF TEXT
430  GOSUB 2000 : REM BIDIRECTIONAL-CONTINUE
     ROUTINE
440  IF DIRECT$="B" THEN GOTO 230
          .
          .
          .
999  END

1000 REM ********** RETURN-TO-CONTINUE ROUTINE
     (FORWARD ONLY)
```

```
1010 VTAB 22
1020 PRINT " (PRESS THE <RETURN> KEY TO
     CONTINUE)"
1030 INPUT NNYTHING$
1040 RETURN : REM FROM RETURN-TO-CONTINUE
     ROUTINE

2000 REM ********** BIDIRECTIONAL-CONTINUE
     ROUTINE
2010 VTAB 22
2020 PRINT " (PRESS F AND <RETURN> TO GO
     FORWARD;"
2030 PRINT " PRESS B AND <RETURN> TO GO BACK)";
2040 INPUT DIRECT$
2050 IF DIRECT$="F" OR DIRECT$="B" THEN GOTO 2130
2055   REM RESPOND TO INVALID RESPONSE:
2060   VTAB 22
2070   PRINT "          (38 blanks)          "
2080   PRINT "          (38 blanks)          "
2090   VTAB 22
2100   PRINT "PLEASE REKEY:"
2110   PRINT " F FOR FORWARD, B FOR BACK";
2120   GOTO 2040
2130 REM END OF RESPONSE VERIFICATION
2140 HOME
2150 RETURN:REM FROM BIDIRECTIONAL-CONTINUE
     ROUTINE
```

You might note that the bidirectional form includes validation of the typed response. See Chapter 8 (RESPONSE VERIFICATION) below for more details on this issue.

5.4 NUMBER OF QUESTIONS

When working with individual students, a teacher often assigns a different number of problems according to each particular student's need. Perhaps you have had students who only feel comfortable working on short assignments and others who enjoy working large numbers of problems.

There are a number of different ways this might arise in CAI. You might want to design the program so that a student is allowed only a certain amount of time with a particular program. That might be wise if computer time is limited and/or if you were interested in students learning to respond rapidly.

Perhaps you are interested in how well the student is doing. Here you could keep track of the percent of questions a student answered correctly. If the student's performance rose above a certain level, the program might move on to the next topic. If, however, the student's percentage fell below a certain level, the program might move to some simpler practice or a review unit of some sort.

In other cases you might want to consider giving the student a limited choice of how many problems to do. This might be appropriate when the computer activity is supplementary to the regular classroom work or when you want students to become more self-directed in their learning.

The latter approach is used for the computer literacy review program for two reasons. The first reason is to permit teachers to direct students to do a random subset of the questions as the final quiz. Our second reason is that this strategy is often both educationally sound and quite easy to implement. The following segment is extracted from the example as it is found in Appendix A.

```
1000 REM
1010 REM  USER INTRODUCTION TO THE PROGRAM:
1020 REM ------------------------------------------------
1030 TEXT
1040 HOME
1050 VTAB 3
1060 PRINT "   I=========================================I"
1070 PRINT "   II                                       II"
1080  PRINT " II     COMPUTER LITERACY REVIEW     II"
1090  PRINT " II                                       II"
1100  PRINT " I=========================================I"
1110  VTAB 10
1120  PRINT " THIS PROGRAM CAN BE USED TO REVIEW"
1130  VTAB 12
1140  PRINT " ALL 20 TERMS AND FACTS OR TO GIVE"
1150  VTAB 14
1160  PRINT " A SHORTER RANDOM QUIZ. IF THIS IS"
1170  VTAB 16
1180  PRINT " TO BE A QUIZ, CHOOSE THE NUMBER OF"
1190  VTAB 18
1200  PRINT " TERMS (1-20) TO DO ACCORDING TO"
1210  VTAB 20
1220  HTAB 4
1230 PRINT "YOUR TEACHER'S INSTRUCTIONS."
1240 VTAB 23
1250 HTAB 12
1260 PRINT "NUMBER OF TERMS";
1270 INPUT NUMBER$
1280 NUMBER=INT ( VAL (NUMBER$) )
1290 REM  RESPONSE VALIDATION
1300 IF NUMBER > 0 AND NUMBER < 21 THEN GOTO
     1340
1310 VTAB 23
1320 PRINT BLANK$
```

```
1330 GOTO 1240
      •      [Later in the program the value NUMBER
      •       is used to choose the number of questions
      •       presented.]
```

One final note is in order here. There is some elementary validation of student responses. The program does not, however, adequately deal with responses such as "5.6" or the word "TWELVE." For more details concerning the validation of student responses, see Chapter 8.

5.5 CHOOSING THE SPEED OF PRESENTATION

In section 2.3 some of the ways in which teachers automatically individualize when working with a single student were noted. A number of ways to individualize CAI have also been considered both in this section and earlier in Chapter 3 (CONTROLLING THE SCREEN). The program segment below combines one of these ideas, that of controlling the speed at which text is displayed on the screen, with the idea of giving students some control over their learning environment. In particular, it provides the student with a choice of the text-presentation speed.

```
80 REM SEGMENT TO PROVIDE USER WITH A CHOICE
85 REM OF TEXT—PRESENTATION SPEED.
90  REM 'ZPEED' IS USED TO REGISTER THE USER'S
    CHOICE
95  HOME
100 VTAB 5
105 PRINT "YOU MAY CHOOSE HOW FAST THE WORDS
    IN"
110 PRINT
115 PRINT "THE LESSON YOU ARE ABOUT TO USE
    APPEAR"
120 PRINT
125 PRINT "ON THE SCREEN:"
130 PRINT
135 SPEED=10
140 PRINT "THIS IS THE SLOW SPEED."
145 PRINT
150 SPEED=120
155 PRINT "THIS IS THE MEDIUM SPEED."
160 PRINT
165 SPEED=255
170 PRINT "THIS IS THE FAST SPEED."
175 PRINT
180 PRINT "TYPE THE NUMBER OF THE SPEED YOU
    WOULD"
```

```
185 PRINT TAB (10)"LIKE TO USE:"
190 PRINT TAB (15)"1. SLOW"
195 PRINT TAB (15)"2. MEDIUM"
200 PRINT TAB (15)"3. FAST"
205 HTAB 20
210 INPUT CHOICE$
215 ZPEED=0
220 IF CHOICE$="1" THEN ZPEED=10
225 IF CHOICE$="2" THEN ZPEED=120
230 IF CHOICE$="3" THEN ZPEED=255
235 IF ZPEED > 0 THEN GOTO 255
240   REM SIMPLE RESPONSE VERIFICATION
245   PRINT "PLEASE REKEY AS 1, 2, OR 3"
250   GOTO 210
255 REM VALID SPEED CHOSEN: SET SPEED AND
    CONTINUE:
260 SPEED=ZPEED
```

•
•
•

[the rest of the program would follow here]

In a similar way, a student can be asked to select the length of pause loops that are included in the program (see section 3.3).

5.6 USER FRIENDLINESS

The term *user friendly* is often used to refer to several desirable features of CAI (and other software too). In many cases this term refers to a program's giving the user meaningful control over the program. In others it means the program has been designed carefully enough that responses are fully checked, and the user never feels trapped or lost (see both Chapter 4 on CHECKING ANSWERS and Chapter 8 on RESPONSE VERIFICATION).

Another program feature often cited as part of user friendliness is the availability of what is called a *help capability*. In its full blown form, such a facility allows the user to type a symbol such as "?" or word such as "HELP" at any point in the program to get instructions or clarifications as to what is expected or what options are possible at that point in the program. Such features are difficult to achieve in CAI, but many of the benefits can be achieved by regularly including some brief instructions as one option in your menus. On the other hand, a program which has narrowly defined goals and a limited number of activities or a program used in a controlled environment (such as with an aide always present) may find little need for such complications.

Keep in mind that your goal in writing programs is not to provide flexibility just for the sake of flexibility. Instead,

imagine yourself working with a variety of students as you design and write your program. Include in your program as many of the things that you naturally do as you can. The more you prepare the computer to teach well, the better assistant it will become for you and your students.

Once the program has been completed, it is especially important that you try it out with students. No matter how carefully you have planned, there will be things that you will discover need to be changed once you have observed the students' performance with the program. Plan your time to allow for these changes before you use the program as an integral part of your instruction.

CAI 6

PROVIDING APPROPRIATE TASKS: CONTROLLING RANDOMIZATION

When you are using a textbook with your students, there are only a limited number of review questions and/or practice problems available for use. When a student needs extra review, you either spend many hours creating or locating new problems, or the student goes without the added practice. The computer can be a powerful aid in this situation.

There are two primary places where randomization is likely to be valuable to you. The first of these concerns the ability of the computer to randomly generate numbers or letters to be used in creating exercises. For example, the computer can generate the two numbers that a student is to add together in a math drill program. Secondly, the computer can randomly pick from a previously created list of questions or options. Thus you might create a long list of questions from which the computer would randomly pick 10 for a short quiz. Along this same line, you can use randomization to choose among a variety of rewards for correct answers so that there is continual variety in the feedback that a student receives.

6.1 RANDOMIZATION AT POWER UP

Let's suppose that you have written a drill-and-practice program which makes use of BASIC's random number generator. Gradually you begin to notice that each time you turn the computer on and your students begin to use the program, they get the same sequence of questions. Your first reaction might be to suspect that your program has a bug.

Actually, that may not be the case. In most cases when you turn on a microcomputer and use its random number generator, you will get the same sequence of random numbers every time. That is certainly not what you had in mind when you used the random number generator! There is usually a quick and easy way around that problem. In Applesoft BASIC, for example, using

the statement

X=RND(−PEEK(78)−256*PEEK(79))

once at the beginning of the program will solve the problem.[1]

6.2 RANDOMIZATION WITH NUMBERS

It is not uncommon to find that two versions of the BASIC language deal with random numbers in slightly different ways. No matter how your version handles these numbers, the general remarks in this section still apply.

If you are making up a worksheet for your students that involves numbers, you set certain criteria for your choice of numbers in the problems. For very young children, you might decide to use numbers less than 8 in a multiplication facts drill. If you were doing addition of fractions with seventh graders, you might decide that no denominator should be bigger than 12. When you create worksheets by hand, you do this almost without thinking. When you are designing a computer program, the thinking needs to be made explicit.

Also, when you have finished your worksheet, you usually work out an answer key. If any of the problems seem ridiculous, you change them. If, however, you write a computer program to generate problems, the computer may create unreasonable problems and produce the answers. The following problem could easily result from a fraction addition drill-and-practice program which does not have thorough problem-evaluation procedures:

6 11/14 + 45 7/39 + 14 6/11

Such a problem is probably not at all what you had in mind.

The point here is that considerable thought needs to go into your plan for problem generation. Sometimes it may be necessary to have the computer examine the problem and perhaps have the program evaluate the answer, just as you would by hand. The computer works so rapidly that the hesitation caused by its having to work a few problems and eliminate unreasonable ones will be so slight that no one will notice.

The actual mechanics of generating random problems is really quite simple. You'll remember that in BASIC there is a RND function that generates random decimals between 0 and 1. Thus, if you are using an Applesoft BASIC, RND(1) returns values like

---

[1]Some other computers use the RANDOMIZE statement. If this is not available either, you can ask the program user to enter a number between 1 and 500 (different each time) and use a FOR/NEXT loop to call the RND function that many times before getting into the program proper.

.443566, .002723, .105834, and .971037.[1] By using a little mathematics and the INT[2] function, you can easily produce random whole numbers of a predefined size range. For example:

```
RNUM=INT(10*RND(1))+1
```

will put some whole number between 1 and 10 (inclusive) in the variable RNUM. Similarly,

```
WHOLE=INT(20*RND(1))
```

will put some whole number between 0 and 19 (inclusive) in the variable WHOLE.

In general, then, the formula

```
X=INT(MAXIMUM*RND(1))+1
```

will put an integer between 1 and MAXIMUM (where MAXIMUM is a positive integer) into the variable X. In similar fashion

```
X=INT(RANGE*RND(1))+START
```

will put an integer between START and START+RANGE−1 in X (again assuming that RANGE and START are integers and that RANGE is positive).

As an example of how these formulas might be used, read the following program segment. It is intended to randomly generate some two-digit addition problems for students to practice. We have also illustrated the idea of problem evaluation by limiting the problems to those whose answers are also two digits.

```
100 REM SEGMENT TO GENERATE RANDOM 2-DIGIT
    ADDITION PROBLEMS
110 REM
120 REM VARIABLES:
130 REM     N1, N2 — 2-DIGIT WHOLE NUMBERS TO BE
            ADDED
140 REM     PBCTR — COUNTS NUMBER OF PROBLEMS
            PRESENTED
150 REM     STSUM — ANSWER KEYED IN BY THE STU-
            DENT
160 REM     SUM — THE SUM OF N1 AND N2
170 REM
200 FOR PBCTR=1 TO 10
210   HOME
```

---

[1] As we indicated earlier, computers often differ in their implementation of the RND function. If you are not using Applesoft BASIC, be sure to find out what kinds of numbers your dialect of BASIC produces.

[2] Recall that INT of a decimal number returns the number itself if it is actually a whole number or the whole number closest to the decimal value and to the left of it on the number line.

```
220   REM CREATE A RANDOM PROBLEM;
230     N1=INT(RND(1)*90)+10
240     N2=INT(RND(1)*90)+10
250     SUM=N1+N2
260     IF SUM > 99 THEN GOTO 220
270   REM PROBLEM IS OK TO PRESENT:
280   VTAB 10:HTAB 12
290   PRINT N1;" + ";N2;" = ";
300   INPUT STSUM
            .                    [code to check the answer and
            .                    provide appropriate feedback]
            .
310 NEXT PBCTR
```

Notice that in lines 230 and 240 the program is generating integers between 10 and 99. In line 250 the computer calculates the correct answer. In line 260 the answer is checked, and if it is larger than 99, the program goes back to generate another problem.

6.3 RANDOMIZATION WITH STRINGS

Creating random questions and answers is not quite so easy in non-mathematical subjects. One way or another, you have to create the questions and appropriate answers. However, once a set of questions of any size is created, it will be more useful in a computer program than it would be on a worksheet. This is because it is relatively easy to present the questions in a random order and to present only a limited number of the available problems.

For this discussion, we will be using the READ/DATA method of entering questions into your program. If your programming progresses to the point that you have large numbers of questions, then, as we have suggested before, you will probably want to learn to use files.

The following steps accomplish randomization with strings:

- Move the questions and answers into arrays.
- Pick a random number between 1 and the number of questions available.
- Use that random number to identify which question to present.

The following segment accomplishes the above tasks. Look especially at lines 310-330 and 500-999.

```
100 REM SEGMENT TO DEMONSTRATE RANDOMIZED
105 REM SELECTION OF PROBLEMS
110 REM
```

```
115 REM VARIABLES USED:
120 REM     ANS$ – ARRAY CONTAINING CORRECT
            ANSWERS
125 REM     CHOICE – RANDOM NUMBER TO SELECT
130 REM     QUESTION TO PRESENT
135 REM     PRESENT – COUNTS PROBLEMS PRESENTED
140 REM     QCTR – COUNTS AVAILABLE QUESTIONS
145 REM     QUES$ – ARRAY CONTAINING THE
            QUESTIONS
150 REM ARRAYS USED IN THE PROGRAM:
155 DIM QUES$(75),ANS$(75)
200 REM DATA STATEMENTS FOR 75 ITEMS:
201 DATA ⟨question 1⟩,⟨answer 1⟩
          •
          •
          •
275 DATA ⟨question 75⟩,⟨answer 75⟩
300 REM LOOP TO READ IN ALL THE QUESTIONS AND
    ANSWERS:
310 FOR QCTR=1 TO 75
320   READ QUES$(QCTR),ANS$(QCTR)
330 NEXT QCTR
500 REM LOOP TO SELECT AND PRESENT 10 RANDOM
    QUESTIONS
510 FOR PRESENT=1 TO 10
520   REM RANDOMLY CHOOSE A QUESTION:
530   CHOICE=INT(75*RND(1))+1
540   PRINT QUES$(CHOICE)
            •          [obtaining and checking of responses
            •                and the reward sections]
            •
999 NEXT PRESENT
```

Another way to accomplish the same task is the following:

- Randomly pick the number of each question to be used.
- Read through the whole list but keep only those whose numbers had been chosen.
- Present the accumulated list.

This reduces the overall memory space needed, but complicates the program code a bit. The following segment illustrates this alternate approach:

```
100 REM SEGMENT TO DEMONSTRATE RANDOMIZED
105 REM SELECTION OF PROBLEMS BY CAPTURING A
110 REM FEW IN A NEW LIST.
115 REM
120 REM VARIABLES USED:
125 REM     ANS$ – ARRAY CONTAINING CORRECT
            ANSWERS
```

```
130 REM     ATEMP$ -- TEMPORARY STORAGE OF
            ANSWER READ
135 REM     DTACTR -- COUNTS THRU COMPLETE DATA
            LIST
140 REM     PRESENT -- COUNTS PROBLEMS
            PRESENTED
145 REM     QCTR -- COUNTS 'CAPTURED' QUESTIONS
150 REM     QTEMP$ -- TEMPORARY STORAGE OF
            QUESTION READ
155 REM     QUES$ -- ARRAY CONTAINING THE CHOSEN
            QUESTIONS
160 REM     WHICH -- ARRAY TO HOLD POSITIONS IN
            THE DATA
165 REM     LIST OF THE 10 ITEMS CHOSEN RANDOMLY
170 REM  ARRAYS USED IN THE PROGRAM
175 DIM WHICH(10),QUES$(10),ANS$(10)
200 REM  DATA STATEMENTS FOR 75 ITEMS
201 DATA (question 1),(answer 1)
        .
        .
        .
275 DATA (question 75),(answer 75)
300 REM  LOOP TO SELECT ITEMS BY POSITION IN THE
    LIST:
310 FOR QCTR=1 TO 10
320   WHICH(QCTR)=INT(75*RND(1)+1)
330 NEXT QCTR
340 REM  LOOP TO READ QUESTIONS AND ANSWERS
350 REM  AND CAPTURE THOSE WHICH WERE SELECTED
360 FOR DTACTR=1 TO 75
370   READ QTEMP$,ATEMP$
380   FOR QCTR=1 TO 10
390     IF WHICH (QCTR) < > DTACTR THEN GOTO 430
400       REM  ELSE CAPTURE THIS ONE:
410       QUES$(QCTR)=QTEMP$
420       ANS$(QCTR)=QANS$
430   NEXT QCTR
440 NEXT DTACTR
500 REM  LOOP TO PRESENT CHOSEN QUESTIONS:
510 FOR PRESENT=1 TO 10
520   VTAB 10
530   PRINT QUES$(PRESENT)
        .                    [obtaining and checking of responses
        .                                and reward sections]
        .
999 NEXT PRESENT
```

When you randomize the selection of questions, it is not uncommon to have the same question come up repeatedly. It is a

fairly simple matter to avoid that problem. Simply check off questions that have been used by putting "YES" in an array USED$. We return to the first of the two examples above to illustrate this idea. The new version has had lines 147, 305, 325 and 532-536 added, and lines 105, 155 and 520 modified when compared to the original.

```
100 REM SEGMENT TO DEMONSTRATE RANDOMIZED
105 REM SELECTION OF PROBLEMS . . . WITHOUT
        REPEATS.
110 REM
115 REM VARIABLES USED;
120 REM     ANS$ — ARRAY CONTAINING CORRECT
            ANSWERS
125 REM     CHOICE — RANDOM NUMBER TO SELECT
130 REM     WHICH QUESTION TO PRESENT
135 REM     PRESENT — COUNTS PROBLEMS
            PRESENTED
140 REM     QCTR - COUNTS AVAILABLE QUESTIONS
145 REM     QUES$ — ARRAY CONTAINING THE
            QUESTIONS
147 REM     USED$ — ARRAY TO MARK QUESTIONS
            ALREADY USED
150 REM ARRAYS USED IN THE PROGRAM:
155 DIM QUES$(75),ANS$(75),USED$(75)
200 REM DATA STATEMENTS FOR 75 ITEMS:
201 DATA ⟨question 1⟩,⟨answer 1⟩
      .
      .
      .
275 DATA ⟨question 75⟩,⟨answer 75⟩
300 REM LOOP TO READ IN ALL THE QUESTIONS AND
305 REM ANSWERS AND TO SET FLAGS:
310 FOR QCTR=1 TO 75
320   READ QUES$(QCTR),ANS$(QCTR)
325   USED$(QCTR)="NO"
330 NEXT QCTR
500 REM LOOP TO SELECT AND PRESENT 10 RANDOM
    QUESTIONS
510 FOR PRESENT=1 TO 10
520   REM RANDOMLY CHOOSE AN UNUSED QUESTION:
530   CHOICE=INT(75*RND(1))+1
532   IF USED$(CHOICE)="YES" THEN GOTO 530
534   REM AN UNUSED ONE HAS BEEN FOUND
536   USED$(CHOICE)="YES"
540   PRINT QUES$(CHOICE)
        .                    [obtaining and checking of responses
        .                          and the reward sections]
        .
999 NEXT PRESENT
```

Notice that if repeats are allowed, there is no reason that the program couldn't ask more questions than are stored. In this case there would necessarily be some repeats, but there would be no guarantee that all the questions would be asked once.

There are other variations on this general theme as well, such as enforced re-asking of questions which were answered incorrectly. This might be best in a spelling or vocabulary program where you would want a question that was missed to be presented again at a later time in the practice session. The idea here is to require additional practice with the questions with which the student had trouble. An example of such logic is provided in Appendix C.

There is also a more subtle issue if you enforce no repeats and expect all of the available questions to be asked. The program can spend inordinate amounts of time randomly seeking the last few unanswered questions. If you encounter this problem, consult the solution provided in the fully worked out computer literacy review program in Appendix A.

If you plan to use randomization in your program design, you can create a program that is much more flexible and more widely usable. Such a program is also more interesting to use than one that always presents the same problems in the same order. On the other hand, careful planning is necessary to be sure that you actually have control over the randomization that you are using.

CAI 7

REWARDS

One of the most visible characteristics of much commercial educational software is its frequent use of attention-catching rewards. Earlier in this booklet the importance of powerful reward structures was discussed. There are many ways that you might put rewards into your programs. The possibilities are only limited by your imagination. You certainly do not need to be bound by presenting only one form of reward for every question that a student answers correctly. In any case, it is important that you design the reward structure as carefully as you do the other aspects of your program.

7.1 CONSIDERATIONS IN CHOOSING AND DESIGNING REWARDS

Rewards for correct answers can vary from simply moving to the next question to extremely flashy graphics and music[1] to small games. Thus, some options are printing out a simple RIGHT, displaying a design or picture formed from text characters (text graphics), or use of multicolor graphics designs. In still other circumstances the student might be given an opportunity to play a game when some particular level of success has been achieved.

The extent and type of rewards used in CAI are quite varied. Some people think that each response should be powerfully rewarded with graphics designs. Others believe that simply verifying that a response is correct is sufficient. Still others say that an adequate reward is provided by merely going on to the next question if the answer given is correct. Such people are recognizing that the student's intrinsic feeling of success can be a powerful motivator, especially with older children. There is little doubt that in the coming years research will give us increasingly definitive answers as to the best type of reward structure to use

---

[1]See Appendix C for a routine to produce a scale using one of the Apple II family of computers.

in various settings. But for now, we will rely on our knowledge of learning theory, our understanding of the student population we are addressing, and our common sense.

Rewarding every response with powerful color graphics may seem great at first, especially as you are just developing your skills in graphics. However, as you think about earlier discussions of learning theory, you can appreciate that such an approach could easily satiate the learner. Further, it was noted that intermittent reinforcement is the most powerful reinforcement schedule. That lends even more credence to the idea that to extensively reward every correct answer is not usually the best path to follow.

At first it may seem impossible to achieve an intermittent reward schedule. If even going on to the next topic or problem is a form of reward, how can some responses not be rewarded? The effect of an intermittent schedule can be achieved, however, by using two or more levels of reward as we will explore in more detail in section 7.3.

Since you are probably planning to write CAI that is tailored specifically to your students, you want to plan carefully around the particular needs of those students. Exactly what you choose to use is up to you, but there are several points that are well worth considering as these choices are being made.

First of all, what are the ages of the students that will be using your program? In general, younger students seem to find colorful graphics more interesting for a longer period of time than do older ones.

Second, what is the environment in which your program will be used? Is it a quiet classroom or library? If so, would sound or music be disruptive to the other activities going on in that location?

Third, how much time does the reward take? If the reward is too long, the student's attention may waver from the learning task at hand. If the reward is too short, it may be missed completely.

Fourth, does the reward actually serve as a reward for the student, or is it something s/he simply has to endure before being allowed to get on with the task at hand? In the latter case it may even generate frustration and anger.

Finally, does the reward structure dominate the program? Does the program give the impression that it was written for the graphics instead of as a teaching tool? One might parenthetically add the caution to not get so wrapped up in the fun of creating rewards that you lose sight of your educational objectives.

The following paragraphs explore some of these concerns and ideas in more detail, keeping in mind the principles of learning theory.

7.2 USING RANDOMIZATION WITH REWARDS

Perhaps you have had an opportunity to use a program for drill and practice which had only one reward for correct answers and discovered just how boring it can become (. . . that is, you got satiated). Such a reward can even cease to be a reward and become a burden. One of the easiest and most effective ways to avoid this pitfall is by the use of randomization.

Are you using a text graphic? Try randomizing its location on the screen or the length of time it lasts. The following brief segment randomly places the word "SPLENDID!" in a rectangle close enough to the center of the screen that it keeps attention there, but large enough to generate some suspense as to where the word will appear next.

```
3040 REM PICK A RANDOM SPOT FOR PRINTING
3050 HSPOT=INT(15*RND(1))+9
3060 VSPOT=INT(10*RND(1))+6
3070 VTAB VSPOT : HTAB HSPOT
3080 PRINT "SPLENDID!"
```

A simple color graphic becomes more captivating if it employs randomly chosen colors. The following few lines produce a randomly colored starburst in the center of the screen.

```
880  FOR CASE=1 TO 2
885    REM CASE1 USES THE RANDOM (NON-BLACK)
     COLOR
890    IF CASE=1 THEN COLOR=INT (15*RND(1))+1
895  REM CASE2 USES BLACK TO ERASE THE STAR
900    IF CASE=2 THEN COLOR=0
905    FOR RADIUS=0 TO 9
910      PLOT 19-RADIUS,19-RADIUS
915      PLOT 19-RADIUS,20+RADIUS
920      PLOT 20+RADIUS,19-RADIUS
925      PLOT 20+RADIUS,20+RADIUS
930    NEXT RADIUS
935  NEXT CASE
```

As you create more and more complicated or catchy rewards, it is natural to want to use them more often. Techniques such as the above can help to keep a good reward fresh and effective for much longer than it could be without randomization.

However, you need to remember that using one reward over and over, in spite of some of the techniques we discussed above, is providing a situation where satiation still can easily occur. If you don't have time to develop several rewards, it would ordinarily be better just to go on to the next problem when an answer is cor-

rect, or quickly flash a simple GOOD or RIGHT on the screen before going on.

A mixture of a few different rewards can easily be accomplished using a segment such as the one below. In this segment the program randomly draws from the three reward routines. It assumes that they are packaged as subroutines which begin at lines numbered 5000, 6000 and 7000. In this example, the subroutine at 5000 is selected 3/6th of the time, the routine at 6000 is selected 2/6th of the time, and the routine at 7000 is selected 1/6th of the time. (See section 7.3)

```
100 REM SUBROUTINE TO RANDOMLY SELECT REWARD
110 CHOICE=INT(6*RND(1))+1
120 ON CHOICE GOSUB 5000,5000,5000,6000,6000,7000
130 RETURN : REM FROM SELECTING (AND
    PRESENTING) REWARD
```

Developing powerful rewards can be time consuming. As you develop and enhance programs, you will gradually accumulate a library of rewards. You should keep these reward segments for possible use in the future.

7.3 USING DIFFERENT LEVELS OF REWARDS

There are at least two important reasons why a single program should be designed to have more than one reward. Probably, the most important single reason is the importance of an intermittent reinforcement schedule as we discussed earlier. The effect of intermittent rewards can be achieved very powerfully if one includes simply going to the next problem, low-key rewards, and more elaborate rewards among the options when a correct answer is given. The example at the end of the preceeding paragraph shows one way to achieve this. If, for example, you placed a powerful reward at 7000, then as we noted, that reward would be selected only one-sixth of the time. Rewards of lesser intensity would be placed at 6000 and 5000 according to the proportion of time you wanted them to be presented.

The second reason for having more than one reward available in one program is related to giving students second or third tries to get an answer correct. Here you might well want to provide the most powerful rewards for getting the correct answer on the first try and less powerful rewards on each successive try. The program code could choose the reward based on the counter or flag which indicates the level of success rather than on a random number. The following example could follow from a section in which NTRIES registers the number of tries it has taken the student to get the correct answer (here as many as three), and where

the reward at line 7000 is the most powerful, the one at line 6000 of intermediate strength, and the one at 5000 the weakest.

```
1450 IF NTRIES=1 THEN GOSUB 7000
1460 IF NTRIES=2 THEN GOSUB 6000
1470 IF NTRIES=3 THEN GOSUB 5000
```

Clearly, combining a random factor along with the concern for level of success would provide an even more varied reward structure. An example of this is in our computer literacy review program in Appendix A.

7.4 LESS OBVIOUS REWARDS

It may be that you most often think of graphics in connection with rewards such as we have discussed above. As you design programs, remember that graphics can also be used to illustrate principles, provide diagrams, and present ideas. Although in these roles their use is not technically that of a reward, they nonetheless enrich the student's experience with the program and in that sense do provide a reward for use of the program.

In fact there are some who argue that graphics should be used *only* when they enhance the topic being taught and not at all otherwise. Whatever your philosophical stance, it is clear that creative and thoughtful use of graphics for teaching can enhance most if not all of the programs that you write for your students.

CAI 8

RESPONSE VERIFICATION

Have you ever run a program and had an experience such as getting a machine error message when you type in an answer; being told the answer is wrong when you know it is right; or having the program ask for a yes-no answer and then do the opposite of your choice? Such experiences are at best frustrating and at worst a disaster. Many of these problems are due to inadequate rather than outright faulty logic. Often they are due to improper verifying of responses. That is, the program author has not taken the time to be sure that the program user actually is giving the answer in the form that is needed.

8.1 EXAMPLES OF RESPONSE CHECKING/ VERIFICATION

One of the most common places where input needs to be checked is after selection of a menu item. For example, a menu might give the choices 1, 2 and 3. Some programmers would provide specific logic for answers of 1, 2 or 3 but not for 4 or 1.5 or any of many other possible responses (whether given by mistake or out of mischief). Exactly what results from the unintended responses depends on the code to handle the input. A segment such as the following can provide for invalid responses by being inserted between accepting the user's answer and branching.

```
650 REM SEGMENT TO SHOW SIMPLE RESPONSE
    VERIFICATION
660 INPUT CHOICE
670 REM BEGIN CHECKING SEGMENT
680 IF (CHOICE=1) OR (CHOICE=2) OR (CHOICE=3)
    THEN GOTO 710
690   PRINT "PLEASE TYPE A 1, 2, OR 3"
700   GOTO 660
710 REM END CHECKING SEGMENT
```

Another place that causes problems is an instance where a *Yes* or *No* answer is called for. Some programs accept just *Y* or *N* and

others require the entire word. Some of the problems can be solved by a segment like the one just given, but a more forgiving strategy is shown in the following segment.

```
 95 REM EXAMPLE WITH MORE VERSATILE RESPONSE
    VERIFICATION
100 INPUT ANS$
105 REM CHECK ANSWER
110 ANS$=LEFT$(ANS$)
115 IF ANS$="Y" OR ANS$="N" THEN GOTO 145
120 REM INVALID ANSWER RECEIVED: INVERSE TO
    DRAW ATTENTION
125   INVERSE
130   PRINT "PLEASE TYPE Y OR N"
135   NORMAL
140   GOTO 100
145 REM VALID ANSWER RECEIVED; CONTINUE.
```

If you try this or examine it closely, you can see that any answer beginning with *Y* is treated like a *Yes*, and any answer beginning with *N* is treated as *No*. Thus, answers like *Yep* and *Nope* will be handled correctly.

A related kind of answer checking occurs in the instance where you want to allow multiple answers or where you must deal with the possibility of both upper and lower case responses. For example, if you asked the question: "Who was the first president of the United States?", your program might have this test:

```
1055 IF ANS$="WASHINGTON" OR ANS$="GEORGE
     WASHINGTON" OR ANS$="Washington" OR ANS$=
     "George Washington" THEN PRINT "GOOD FOR YOU!"
```

As you progress in your programming, you will develop a number of response verifying routines that are appropriate to your subject area and/or programming style. It's a good idea to write a skeleton program, that is one consisting of a collection of subroutines that you frequently use. Prominent among these would be your response-verification routines. Then when you are ready to begin a new program, starting with this skeleton not only saves you typing, but provides code that is already debugged. To see how a routine such as the one just given can be repackaged to function as a subroutine, consider the following main segment with subroutine.

```
145 REM SEGMENT FROM MAIN PROGRAM
150 INPUT ANS$
155 GOSUB 5000
160 REM GO ON TO PROCESS ANS$
```

```
5000 REM ********** ANS$ CHECKING SUBROUTINE
5005 ANS$=LEFT$(ANS$)
5010 IF ANS$="Y" OR ANS$="N" THEN GOTO 5040
5015 REM INVALID ANSWER PROCESSING:
5020   INVERSE
5025   PRINT "PLEASE TYPE Y OR N"
5028   NORMAL
5030   INPUT ANS$
5035   GOTO 5000
5040 RETURN : REM FROM ANS$ CHECK ROUTINE
```

Note that the subroutine needs its own INPUT ANS$ statement, since it cannot go back to the calling location (remember there might be many of them).

8.2 THE "REENTER" PROBLEM

Have you ever used a program requiring a numeric answer and unwittingly typed in a letter? Think back to the first time this happened to you, and you'll probably remember being completely bewildered; the computer said, "REENTER," and each time you did re-enter your answer it said, 'REENTER" again. Students too are mystified by the less-than-clear message that Applesoft BASIC provides in such an instance. By the time a student asks for help, the original question may have scrolled off the screen because of many unsuccessful attempts to comply with the RE-ENTER directive. Without being able to see the question, even the teacher cannot remedy the situation. This is therefore a point at which verifying responses is quite important.

If the answer is numeric only for convenience, such as in a menu, there is little need to process it as a numeric variable. Use a string variable instead, and check the response as described in the previous section. You need not even lose the ability to use ON GOTO and ON GOSUB statements, because after the response has been checked, it can be converted to its numeric counterpart. The following segment briefly illustrates this for a menu having choices 1, 2 and 3.

```
3300 REM WHILE LOOP TO GET VALID STRING ANSWER
3310   INPUT NUM$
3320   IF NUM$="1" OR NUM$="2" OR NUM$="3" THEN
       GOTO 3360
3330   REM RESPONSE WHEN NOT AN ACCEPTABLE
           ANSWER
3340   PRINT "REKEY AS 1, 2, OR 3"
3350   GOTO 3300
3360 REM    VALID ANSWER; CONVERT TO NUMERIC
           AND CONTINUE:
```

```
3370 NUM=VAL(NUM$)
3380 ON NUM GOSUB 4000,4500,4600
```

If, however, the response is more inherently numeric, such as the answer to an arithmetic problem, a more complicated algorithm is needed. The segment given below will check for whole number answers. Further tests need to be added if you wish to allow numbers with decimals (.), plus (+) or minus (−) signs, or exponential notation (E).

The heart of the algorithm contained in the segment below consists of checking each character of the response to be sure it is a numeric character and setting a flag if any non-numeric character is detected.

```
100 REM ******* NUMERIC RESPONSE CHECKING
                SUBROUTINE
110 REM VARIABLES:
112 REM     IN$—'INPUT' NUMBER STRING
114 REM     CHCTR—COUNTS CHARACTERS IN IN$
116 REM     MARKER$—FLAG TO INDICATE NUMBER/
            NON-NUMBER
118 REM     OUT—RESULTING NUMBER
120 REM     TRY$—CHARACTERS BEING CHECKED
140 REM BEGIN CHECKING:
150 MARKER$="NUMBER"
160 REM SPECIAL CASE: CARRIAGE RETURN=NOT
        NUMERIC
170 IF IN$ < > " " THEN GOTO 200
180   MARKER$="NONNUMBER"
190   GOTO 290
200 REM BEGIN MAIN VERIFICATION 'WHILE' LOOP
210 FOR CHCTR=1 TO LEN (IN$)
220   TRY$=MID$(IN$,CHCTR,1)
230   IF ASC(TRY$) > 47 and ASC(TRY$) < 58 THEN GOTO
      280
240     REM ELSE RESULT IS NONNUMBER
250     MARKER$="NONNUMBER"
260     REM FORCE END OF FOR-NEXT LOOP
270     CHCTR=LEN(IN$)
280 NEXT CHCTR
290 REM MAIN 'WHILE' LOOP DONE: RESPOND TO
        RESULT:
300 IF MARKER$="NUMBER" THEN GOTO 350
310   REM ELSE IT'S NON-NUMERIC; ASK FOR
          RE-ENTRY:
320   PRINT "THAT'S NOT A NUMBER, PLEASE REKEY
      IT:"
330   INPUT IN$
340   GOTO 140
```

```
350 REM  IN$ IS NOW OK; CONVERT TO NUMERIC:
360 OUT=VAL(IN$)
370 RETURN : REM FROM VERIFICATION ROUTINE
```

8.3 SPECIAL CASES

There is one class of circumstances in which something similar to verifying responses is also advisable. When the users are asked about some pivotal action such as replacing a set of data they have previously typed in or going back to the very beginning of a long section, it is generous to confirm the choice before actually carrying it out. The following segment illustrates this idea.

```
765 PRINT "RETURN TO BEGINNING?"
770 INPUT YN$ : GOSUB 5000
775 REM GOSUB AT 5000 IS RESPONSE CHECK FOR Y/N
780 IF YN$="N" THEN GOTO 800
783 REM USER HAS SAID 'YES' -- BE SURE HE MEANS IT
785   PRINT "ARE YOU  S U R E ?"
790   INPUT YN$
793   GOSUB 5000
795   IF YN$="Y" THEN GOTO (line number at the
      beginning)
800 REM CONTINUING. . .
```

As the examples above illustrate, verifying responses is not always simple, and complete checking of all responses may not always be feasible. However, many problems can be avoided by adhering to the following principles.

- Use a consistent form for the answers you expect in a given program.
- Make sure your directions are complete and specific.
- Be willing to take the programming time and space to check all input at an appropriate level.
- Observe your program in actual use; be willing to revise it in the light of students' actual problems.
- Anticipate as many student errors as you can at the point of program design.

CAI 9

POSTSCRIPT

This brings this booklet to a close. You no doubt realize that many of the issues discussed here could be treated in much more depth. Further, some of the concerns discussed have probably raised questions in your mind that have not been completely answered in this booklet. While we have not attempted to provide a definitive treatment of all aspects of CAI, we hope to have provided you with an overview of the issues and some suggestions as to answers you might be able to use.

There are two general areas that you might be interested in exploring in more depth. The first of these is the general consideration of program and/or curriculum design. This booklet has focused primarily on individual programs rather than on groups of programs tied together to form a coherent curriculum. Some different concerns must be attended to if an entire curriculum is being written. For more treatment of this area you could explore textbooks on learning theory, programmed learning, and design of CAI.

The second area that you might want to explore in more depth is the art of programming itself. This soon becomes more specifically machine-oriented than much of this booklet has been. As we have mentioned several times, the use of files is an important step in increasing the power of your programs. For this you need to look for a book that goes beyond beginning BASIC. If you are working with Applesoft BASIC you might want to further explore such things as using ONERR codes to verify input, POKES to set text windows, built-in ROM routines, high resolution graphics and shape tables, or even machine or assembler language programming. The more you develop your programming skill, the more sophisticated your programs will become.

As a parting thought, we would like to encourage you to share your experiences with others. This can range from informal conversations with colleagues; to participation in local, state and national professional meetings; to preparing articles for various computer-related periodicals; to trying to market your CAI creations. Not only will you enrich others' understanding of CAI, but you will learn from and be stimulated by them as well.

APPENDIX A:
COMPUTER LITERACY
REVIEW PROGRAM

```
100  REM — REVIEW-QUIZ ON SELECTED COMPUTER
         FACTS AND TERMS —
110  REM (SAMPLE CAI PROGRAM DEMONSTRATING
120  REM  SOME OF THE CONCEPTS DEVELOPED IN
130  REM  THIS BOOKLET)
140  REM VARIABLE LIST (WITH ASSOCIATED 'DIMS'):
150  REM ..... FOR MAIN PROGRAM:
160  REM ANS$—ARRAY OF ALL AVAILABLE CORRECT
         ANSWERS
170  DIM ANS$(20)
180  REM BLANK$—STRING CONSTANT FOR BLANK LINE
190  REM CHOICE—SUBSCRIPT FOR CHOOSING
         QUESTIONS TO BE USED
200  REM FRIGHT—NUMBER RIGHT ON THE FIRST TRY
210  REM LINECTR—LOOP INDEX FOR PARTIAL SCREEN
         CLEARING
220  REM NUMBER—INTEGER NUMBER OF PROBLEMS TO
         DO
230  REM NUMBER$—USER INPUT OF NUMBER TO DO
240  REM PAIRCTR—COUNTS PAIRS READ IN
250  REM PAUSESIZE—NUMBER OF CYCLES TO USE IN
         PAUSE ROUTINE LOOP
260  REM P1ART$, P2ART$—TO SPLIT QUESTIONS FOR
         CENTERING ON THE SCREEN
270  REM QUES$—ARRAY OF ALL AVAILABLE
         DEFINITIONS
280  DIM QUES$(20)
290  REM QUESCTR—COUNTS QUESTIONS PRESENTED
300  REM RIGHT—NUMBER RIGHT
310  REM SANS$—ANSWER INPUT BY STUDENT
320  REM SRIGHT—NUMBER RIGHT ON THE SECOND TRY
330  REM TRYCTR—COUNTS ATTEMPTS AT A PROBLEM
340  REM UANS$—ARRAY OF ANSWERS TO BE USED
350  DIM UANS$(20)
360  REM UQUES$—ARRAY OF QUESTIONS TO BE USED
```

```
370   DIM UQUES$(20)
380   REM  USED$—ARRAY TO INDICATE WHICH
           QUESTIONS HAVE BEEN CHOSEN
390   DIM USED$(20)
400   REM  . . . . . FOR SUBROUTINES ONLY:
410   REM  CLR—COLOR IN GRAPHICS DESIGN
420   DIM CLR(13)
430   REM  COUNT—COUNTER FOR 'GREAT' LOOP
440   REM  CTR—TO CONTROL NUMBER OF LINES THE
           SAME COLOR
450   REM  CYCLE—LOOP INDEX FOR PAUSE ROUTINE
460   REM  NNYTHING$—HOLDS RETURN MESSAGE ON
           THE SCREEN
470   REM  ROW—ROW TO PLOT IN GRAPHICS
480   REM  TIMES—USED FOR COLOR BLOCKS IN GRAPHICS
           REWARD
490   REM  WHICH—USED TO CHOOSE WHICH REWARD
500   REM  WRD—PARTICULAR REWARD TO BE USED IN
           REWARD #4
510   REM  WRD$—WORDS USED IN REWARD #4
520   REM  X—USED IN REWARD #1 FOR X COORDINATES
530   REM  Y—USED IN REWARD #1 FOR Y COORDINATES
600   REM
610   REM  ———————— INITIALIZATIONS PRIOR TO 'BEGIN'
620   TEXT
630   HOME
640 BLANK$="                    (40 blanks)                    "
650   REM  —TO START THE RANDOM NUMBER
           GENERATOR RANDOMLY:
660   X=RND (—PEEK (78)—256 * PEEK (79))
670   REM  —ESTABLISH 20 QUESTIONS, ANSWERS, AND
            USE STATUS:
680   FOR PAIRCTR=1 TO 20
690   : : : READ QUES$(PAIRCTR),ANS$(PAIRCTR)
700   : : :USED$(PAIRCTR)="NOT USED"
710   NEXT PAIRCTR
720   DATA CHIP IS A COMMON TERM FOR AN
      INTEGRATED _____,CIRCUIT
730   DATA THE LETTER OFTEN USED TO DENOTE
      APPROXIMATELY 1000 IS _____,K
740   DATA CPU STANDS FOR CENTRAL _____
      UNIT,PROCESSING
750   DATA THE TERM 'IC' STANDS FOR THE WORDS
      _____,INTEGRATED CIRCUIT
760   DATA THE WORD 'BIT' STANDS FOR BINARY
      _____,DIGIT
770   DATA IN MOST COMPUTERS 1 BYTE CONSISTS OF
      HOW MANY BITS?,8
780   DATA THE TERM 'VLSI' STANDS FOR THE WORDS
      _____,VERY LARGE SCALE INTEGRATION
```

```
790   DATA THE LAST NAME OF THE DESIGNER OF THE
      'ANALYTICAL ENGINE' IS _____,BABBAGE
800   DATA WHAT PREFIX IS USED TO DESIGNATE ONE
      MILLION?,MEGA
810   DATA THE TERM 'ROM' STANDS FOR THE WORDS
      _____,READ ONLY MEMORY
820   DATA A NANOSECOND IS ONE _____ OF A
      SECOND BILLIONTH
830 DATA  WHO USED HOLES IN CARDS FOR THE 1890
      CENSUS? (LAST NAME ONLY),HOLLERITH
840   DATA THE LAST NAME OF THE FIRST COMPUTER
      PROGRAMMER IS _____,LOVELACE
850   DATA WHO INVENTED A LOOM WHICH STORED
      PATTERNS ON CARDS? (LAST NAME ONLY),JACQUARD
860   DATA THE THREE LETTER COMMON NAME FOR A
      MISTAKE IN A COMPUTER PROGRAM IS _____,
      BUG
870   DATA THE TERM 'RAM' STANDS FOR _____?,
      RANDOM ACCESS MEMORY
880   DATA WHO INVENTED THE FIRST MECHANICAL
      CALCULATOR? (LAST NAME ONLY),PASCAL
890   DATA THE COMPUTER LANGUAGE DEVELOPED IN
      THE 1960'S AT DARTMOUTH IS _____,BASIC
900   DATA THE COMPUTER LANGUAGE DEVELOPED BY
      PAPERT FOR CHILDREN IS _____,LOGO
910   DATA THE FIRST ELECTRONIC COMPUTER BUILT IN
      THE U.S. WAS NAMED _____,ENIAC
1000  REM
1010  REM  USER INTRODUCTION TO THE PROGRAM:
1020  REM  ------------------------------------------------
1030  TEXT
1040  HOME
1050  VTAB 3
1060  PRINT "    I==========================I"
1070  PRINT "    II                         II"
1080  PRINT "    II     COMPUTER LITERACY REVIEW    II"
1090  PRINT "    II                         II"
1100  PRINT "    I==========================I"
1110  VTAB 10
1120  PRINT "    THIS PROGRAM CAN BE USED TO REVIEW"
1130  VTAB 12
1140  PRINT "    ALL 20 TERMS AND FACTS OR TO GIVE"
1150  VTAB 14
1160  PRINT "    A SHORTER RANDOM QUIZ. IF THIS IS"
1170  VTAB 16
1180  PRINT "    TO BE A QUIZ, CHOOSE THE NUMBER OF"
1190  VTAB 18
```

*If this program is to be entered and run, then three blanks should be left after the beginning quote and 32 "=" signs should be used to accomplish the desired spacing.

```
1200  PRINT "    TERMS (1-20) TO DO ACCORDING TO"
1210  VTAB 20
1220  HTAB 4
1230   PRINT "YOUR TEACHER'S INSTRUCTIONS."
1240  VTAB 23
1250  HTAB 12
1260  PRINT "NUMBER OF TERMS";
1270  INPUT NUMBER$
1280  NUMBER=INT ( VAL (NUMBER$) )
1290  REM  RESPONSE VALIDATION
1300  IF NUMBER > 0 AND NUMBER < 21 THEN GOTO 1340
1310  VTAB 23
1320  PRINT BLANK$
1330  GOTO 1240
1340  REM ----------
1350  REM  SELECTION OF QUESTIONS TO ACTUALLY
           PRESENT-
1360  REM  -FIRST, RANDOM SELECTION OF ALL EXCEPT
           LAST TWO:
1370  FOR QUESCTR=1 TO (NUMBER-2)
1380  : : :CHOICE=INT (20 * RND (1) )+1
1390  : : : REM  CHECK IF HAS BEEN SELECTED
1400  : : : IF USED$(CHOICE)="CHOSEN" THEN GOTO 1380
1410  : : :UQUES$(QUESCTR)=QUES$(CHOICE)
1420  : : :UANS$(QUESCTR)=ANS$(CHOICE)
1430  : : :USED$(CHOICE)="CHOSEN"
1440  NEXT QUESCTR
1450  QUESCTR=NUMBER-2
1460  REM  -THEN CHOOSE LAST 2 NON-RANDOMLY:
1470  FOR CHOICE=1 TO 20
1480  : : : IF USED$(CHOICE)="CHOSEN" THEN GOTO 1560
1490  : : : : : : REM  HERE'S AN AS-YET UNCHOSEN ONE:
1500  : : : : :QUESCTR=QUESCTR+1
1510  : : : : :UQUES$(QUESCTR)=QUES$(CHOICE)
1520  : : : : :UANS$(QUESCTR)=ANS$(CHOICE)
1530  : : : : :USED$(CHOICE)="CHOSEN"
1540  : : : : : : REM  -IF IT'S THE LAST NEEDED, FORCE
                     END OF FOR-NEXT LOOP:
1550  : : : : : : IF QUESCTR >= NUMBER THEN CHOICE=20
1560  NEXT CHOICE
2000  REM
2010  REM  MAIN PROGRAM SEGMENT:
2020  REM ----------
2030  REM  -PRESENT QUESTIONS ONE AT A TIME-
2040  RIGHT=0
2050  FRIGHT=0
2060  SRIGHT=0
2070  FOR QUESCTR=1 TO NUMBER
2080  : : : HOME
2090  : : : REM  -CENTER AND PRINT QUESTION
2100  : : :P1ART$=" "
2110  : : :P2ART$=" "
```

```
2120  : : : REM --'IF-THEN-ELSE' IN CASE QUESTION NEEDS
                2 LINES:
2130  : : : IF LEN (UQUES$(QUESCTR) ) < 41 THEN GOTO 2230
2140  : : : REM --QUESTION NEEDS 2 LINES; FIND BLANK
                TO END FIRST LINE:
2150  : : : : : :CTR=40
2160  : : : : : : IF MID$ (UQUES$(QUESCTR),CTR,1)=" "
THEN GOTO 2200
2170  : : : : : : : : :CTR=CTR-1
2180  : : : : : : : : : GOTO 2160
2190  : : : : : : REM --SPLIT INTO TWO PARTS:
2200  : : : : : :P1ART$=MID$ (UQUES$(QUESCTR),1,CTR)
2210  : : : : : :P2ART$=MID$ (UQUES$(QUESCTR),CTR+1, LEN
(UQUES$(QUESCTR) ) -CTR)
2220  : : : : : : GOTO 2250
2230  : : : REM --ELSE QUESTION FITS ON ONE LINE:
2240  : : : : : :P2ART$=UQUES$(QUESCTR)
2250  : : : REM --END OF 'IF-THEN-ELSE' FOR SPLITTING
                QUESTION
2260  : : : VTAB 8
2270  : : : HTAB 1+( (40-LEN (P1ART$) ) / 2)
2280  : : : PRINT P1ART$
2290  : : : VTAB 9
2300  : : : HTAB 1+( (40-LEN (P2ART$) ) / 2)
2310  : : : PRINT P2ART$
2320  : : : VTAB 11
2330  : : : HTAB 5
2340  : : : PRINT "YOUR ANSWER IS --> ";
2350  : : : INPUT SANS$
2360  : : :TRYCTR=1
2370  : : : REM .....
2380  : : : REM 'WHILE LOOP' TO CHECK ANSWER AND
                RESPOND:
2390  : : : IF SANS$=UANS$(QUESCTR) OR TRYCTR > 2
THEN GOTO 2710
2400  : : : : : : REM --'IF-THEN-ELSE' TO CHECK WHICH
                    TRY:
2410  : : : : : : IF TRYCTR < > 1 THEN GOTO 2580
2420  : : : : : : REM --FIRST TRY WAS WRONG:
2430  : : : : : : : : : VTAB 14
2440  : : : : : : : : : HTAB 14
2450  : : : : : : : : : PRINT "NO, TRY AGAIN!"
2460  : : : : : : : : : REM --RETURN-TO-CONTINUE:
2470  : : : : : : : : : GOSUB 40000
2480  : : : : : : : : : VTAB 11
2490  : : : : : : : : : FOR LINECTR=1 TO 13
2500  : : : : : : : : : : : PRINT BLANK$
2510  : : : : : : : : : NEXT LINECTR
2520  : : : : : : : : : VTAB 11
2530  : : : : : : : : : HTAB 5
2540  : : : PRINT "YOUR ANSWER IS --> ";
```

```
2550 : : : : : : : : : INPUT SANS$
2560 : : : : : : : : : :TRYCTR=2
2570 : : : : : : : : : GOTO 2690
2580 : : : : : : REM —SECOND TRY WAS WRONG:
2590 : : : : : : : : : VTAB 15
2600 : : : : : : : : : HTAB 8
2610 : : : : : : : : : PRINT "NO, THE CORRECT ANSWER IS:"
2620 : : : : : : : : : PRINT
2630 : : : : : : : : : REM —CENTER AND PRINT CORRECT
                          ANSWER:
2640 : : : : : : : : : HTAB ( (40—LEN (UANS$(QUESCTR) ) ) / 2)
2650 : : : : : : : : : PRINT UANS$(QUESCTR)
2660 : : : : : : : : : REM —RETURN-TO-CONTINUE:
2670 : : : : : : : : : GOSUB 40000
2680 : : : : : : : : : :TRYCTR=3
2690 : : : : : : REM —END OF IF-THEN-ELSE TO CHECK
                        WHICH TRY
2700 : : : : : : GOTO 2370
2710 : : : REM —COUNT NUMBER RIGHT:
2720 : : : IF TRYCTR=1 THEN FRIGHT=FRIGHT+1
2730 : : : IF TRYCTR=2 THEN SRIGHT=SRIGHT+1
2740 : : : REM .....
2750 : : : REM —END OF WHILE LOOP FOR ANSWER
                PROCESSING
2760 : : : REM —PROVIDE REWARD IF QUESTION WAS
                ANSWERED CORRECTLY:
2770 : : : IF TRYCTR < 3 THEN GOSUB 10000
2780 NEXT QUESCTR
3000 REM
3010 REM———————————————————————————————
3020 REM REPORT OF RESULTS:
3030 HOME
3040 VTAB 10
3050 RIGHT=FRIGHT+SRIGHT
3060 PRINT "YOU GOT ";RIGHT;" OUT OF ";NUMBER;
     " CORRECT:"
3070 PRINT
3080 PRINT "⟨10 blanks⟩";FRIGHT ;" ON THE SECOND TRY."
3090 PRINT
3100 PRINT "⟨10 blanks⟩";SRIGHT;" ON THE SECOND TRY."
3110 VTAB 23
3120 END
3130 REM == END OF PROGRAM == END OF PROGRAM ==
3140 REM
10000 REM ***************************************************
10010 REM SUBROUTINE FOR SELECTION OF REWARDS:
10020 ON TRYCTR GOSUB 20000,30000
10030 RETURN
10040 REM **** END OF REWARD SELECTION ROUTINE.
20000 REM ***************************************************
```

```
20010 REM REWARD SELECTION SUBROUTINE FOR
          CORRECT-ON-FIRST-TRY:
20020 WHICH=INT (3 * RND (1) )+1
20030 ON WHICH GOSUB 21000,22000,23000
20040 RETURN
20050 REM **** END OF 1ST TRY REWARD SELECTION
          ROUTINE.
21000 REM **********************
21010 REM COLOR GRAPHICS REWARD SUBROUTINE:
21020 CLR(0)=11
21030 CLR(1)=1
21040 CLR(2)=9
21050 CLR(3)=8
21060 CLR(4)=13
21070 CLR(5)=14
21080 CLR(6)=12
21090 CLR(7)=4
21100 CLR(8)=7
21110 CLR(9)=6
21120 CLR(10)=2
21130 CLR(11)=3
21140 CLR(12)=10
21150 HOME
21160 GR
21170 FOR TIMES=0 TO 12
21180 : : : COLOR=CLR(TIMES)
21190 : : : FOR ROW=3 * TIMES TO 3 * TIMES+3
21200 : : : : : : HLIN TIMES,TIMES+5 AT ROW
21210 : : : : : : HLIN (39-TIMES),(39-5-TIMES) AT ROW
21220 : : : NEXT ROW
21230 NEXT TIMES
21240 FOR CTR=36 TO 39
21250 : : : HLIN 13,27 AT CTR
21260 NEXT CTR
21270 COLOR=0
21280 HLIN 14,25 AT 38
21290 HLIN 14,25 AT 39
21300 VTAB 21
21310 HTAB 16
21320 PRINT "R I G H T ! "
21330 REM -INVOKE PAUSE SUBROUTINE:
21340 PAUSESIZE=500
21350 GOSUB 41000
21360 TEXT
21370 HOME
21380 RETURN
21390 REM **** END OF GRAPHICS REWARD ROUTINE.
22000 REM ************************
22010 REM SUBROUTINE FOR TEXT REWARD #1 (WORDS IN
          PARABOLIC ARCH):
22020 HOME
```

```
22030 FOR X=5 TO 35
22040 : : :Y=( - (X-20)^2) / 16) + 1
22050 : : : VTAB (Y+3)
22060 : : : HTAB X
22070 : : : PRINT "SUPER!"
22080 NEXT X
22090 REM -INVOKE PAUSE SUBROUTINE:
22100 PAUSESIZE=300
22110 GOSUB 41000
22120 HOME
22130 RETURN
22140 REM ****END OF TEXT REWARD #1 ROUTINE.
23000 REM **********************
23010 REM SUBROUTINE FOR TEXT REWARD #2
        (RANDOMLY PLACED 'GREAT'S):
23020 HOME
23030 FOR COUNT=1 TO 35
23040 : : :X = INT ( RND (1) * 34) + 1
23050 : : :Y = INT ( RND (1) * 23) + 1
23060 : : : VTAB Y
23070 : : : HTAB X
23080 : : : PRINT "GREAT!"
23090 NEXT COUNT
23100 REM -INVOKE PAUSE SUBROUTINE:
23110 PAUSESIZE=400
23120 GOSUB 41000
23130 HOME
23140 RETURN
23150 REM **** END OF TEXT REWARD #2 ROUTINE.
30000 REM ******************************************************
30010 REM REWARD SELECTION SUBROUTINE FOR
        CORRECT-ON-SECOND TRY:
30020 WHICH=INT ( RND (1) * 3) + 1
30030 ON WHICH GOSUB 31000,32000,32000
30040 RETURN
30050 REM *** END OF 2ND TRY REWARD CHOICE ROUTINE.
31000 REM **********************
31010 REM SUBROUTINE FOR TEXT REWARD #3 (BIG Y-E-S):
31020 HOME
31030 VTAB 8
31040 PRINT "          Y     Y  EEEE  SSSS"
31050 PRINT "           Y   Y   E     S"
31060 PRINT "            Y Y    E     S"
31070 PRINT "             Y     EEE   SSSS"
31080 PRINT "             Y     E        S"
31090 PRINT "             Y     E        S"
31100 PRINT "             Y     EEEE  SSSS"
31110 REM -INVOKE PAUSE SUBROUTINE:
31120 PAUSESIZE=500
31130 GOSUB 41000
31140 HOME
```

```
31150 RETURN
31160 REM **** END OF TEXT REWARD #3 ROUTINE.
32000 REM **********
32010 REM SUBROUTINE FOR TEXT REWARD #4 (RANDOM
      WORD):
32020 REM (ASSIGNMENT STATEMENTS USED TO KEEP
      SEGMENTS INDEPENDENT: AVOIDS NEED FOR
32030 REM RESTORES OR GROUPED READ/DATA
      STATEMENTS.)
32040 WRD$(1)="RIGHT!"
32050 WRD$(2)="GOOD!"
32060 WRD$(3)="O.K.!"
32070 WRD$(4)="GOT IT"
32080 WRD$(5)="CORRECT"
32090 WRD$(6)="FINE!"
32100 WRD=INT ( RND (1) * 6) + 1
32110 VTAB 14
32120 PRINT BLANK$
32130 VTAB 14
32140 HTAB 16
32150 FLASH
32160 PRINT WRD$(WRD)
32170 REM --INVOKE PAUSE SUBROUTINE:
32180 PAUSESIZE=750
32190 GOSUB 41000
32200 NORMAL
32210 HOME
32220 RETURN
32230 REM **** END OF TEXT REWARD #4 ROUTINE.
40000 REM **************************************************
40010 REM RETURN-TO-CONTINUE SUBROUTINE:
40020 VTAB 23
40030 PRINT "  ( PRESS RETURN KEY TO CONTINUE )  ";
40040 INPUT NNYTHING$
40050 RETURN
40060 REM **** END OF RETURN-TO-CONTINUE ROUTINE.
41000 REM **************************************************
41010 REM PAUSE SUBROUTINE:
41020 : : FOR CYCLE=1 TO PAUSESIZE
41030 NEXT CYCLE
41040 RETURN
41050 REM **** END OF PAUSE ROUTINE.
42000 REM ===== END OF ALL SUBROUTINES =====
42010 REM
42020 REM **** END OF APPENDIX A ****
```

APPENDIX B:
"CASE" STRUCTURE FOR MULTIPLE TRIES

```
100 REM THIS PROGRAM DEMONSTRATES A STRUCTURE
110 REM  THAT ALLOWS THE STUDENT MULTIPLE TRIES
120 REM  ON A QUESTION, GIVES THE STUDENT MULTIPLE
130 REM  HINTS, AND GIVES MULTIPLE LEVEL REWARDS!
132 REM  IT ALSO IS AN EXAMPLE OF HOW TO
         IMPLEMENT A 'CASE' STATEMENT IN BASIC.
150 REM VARIABLE LIST:
160 REM CANS$ -- CORRECT ANSWER
170 REM FLAG$ -- TO IDENTIFY THE ATTEMPT AT
         PROBLEM
180 REM SANS$ -- STUDENT'S ANSWER
190 REM
200 REM BEGIN PROGRAM
210 REM INITIALIZE FLAG$
220 FLAG$="FIRST"
230 HOME
240 VTAB 10: HTAB 10
250 REM PRESENT QUESTION HERE
270 REM FREEZE THE TOP 12 LINES OF THE SCREEN:
280 REM ('HOME' WILL THEN CLEAR ONLY BELOW THE
         12TH LINE)
290 POKE 34,12
300 INPUT SANS$
310 PRINT
320 REM ********** 'WHILE' LOOP TO CHECK ANSWER:
330 IF SANS$=CANS$ THEN GOTO 820
340 : : : REM
350 : : : REM ********** CASE STATEMENT TO CHECK
         WHICH TRY
360 : : : REM (A CASE IS A MULTIPLE IF-THEN-ELSE)
370 : : : IF FLAG$ < > "FIRST" THEN GOTO 440
380 : : : REM CASE OF FIRST TRY
390 : : : : : : : REM INSERT FIRST SUGGESTION HERE
400 : : : : : : PRINT "NO, TRY AGAIN!"
410 : : : : : : INPUT SANS$
420 : : : : : :FLAG$="SECOND"
```

```
430  : : : : : : GOTO 780
440  : : : REM CASE OF SECOND TRY
450  : : : IF FLAG$ < > "SECOND" THEN GOTO 510
460  : : : : : : REM  INSERT SECOND SUGGESTION HERE
470  : : : : : : PRINT "NO, TRY AGAIN!"
480  : : : : : : INPUT SANS$
490  : : : : : : FLAG$="THIRD"
500  : : : : : : GOTO 780
510  : : : REM CASE OF THIRD TRY
520  : : : IF FLAG$ < > "THIRD" THEN GOTO 580
530  : : : : : : REM  INSERT THIRD SUGGESTION HERE
540  : : : : : : PRINT "NO, TRY AGAIN!"
550  : : : : : : INPUT SANS$
560  : : : : : :FLAG$="NTH"
570  : : : : : : GOTO 780
580  REM       •
590  REM       •
600  REM       •                          [Any number of tries here]
610  REM       •
620  REM       •
630  : : : IF FLAG$ < > "NTH" THEN GOTO 700
640  : : : REM CASE OF NTH TRY
650  : : : : : : REM  INSERT NTH SUGGESTION HERE
660  : : : : : : PRINT "NO, TRY AGAIN!"
670  : : : : : : INPUT SANS$
680  : : : : : :FLAG$="LAST"
690  : : : : : : GOTO 780
700  : : : REM CASE OF LAST TRY
710  : : : IF FLAG$ < > "LAST" THEN GOTO 780
720  : : : : : : REM CORRECT ANSWER GIVEN
730  : : : : : : PRINT "NO, THE CORRECT ANSWER IS . . . "
740  : : : : : :FLAG$="DONE"
750  : : : : : : REM (IF YOU WANT TO KEEP THE STUDENT
760  : : : : : : REM  ANSWER, YOU MAY CHOOSE TO EXIT
                      DIFFERENTLY THAN IN 770)
770  : : : : : :SAN$=CANS$
780  : : : REM ------END CASE TO CHECK WHICH TRY
800  : : : PRINT
805  : : : PRINT
810  : : : GOTO 320
820  REM ******* END OF 'WHILE' LOOP TO CHECK ANSWER
830  REM
840  REM  PUT THE SCREEN BACK TO NORMAL SCROLLING:
850  POKE 34,0
860  REM
870  REM $$$$$$$$ CASE TO DETERMINE LEVEL OF REWARD
880  REM CASE OF FIRST TRY
890  IF FLAG$ < > "FIRST" THEN GOTO 930
900  : : : PRINT "FANTASTIC!"
910  : : : GOTO 1080
920  REM CASE OF SECOND TRY
```

```
930  IF FLAG$ < > "SECOND" THEN GOTO 960
940  : : : PRINT "GOOD FOR YOU!"
950  : : : GOTO 1080
960  REM CASE OF THIRD TRY
970  IF FLAG$ < > "THIRD" THEN GOTO 1050
980  : : : PRINT "FINE"
990  : : : GOTO 1080
1000 REM      •
1010 REM      •
1020 REM      •                          [any number of rewards]
1030 REM      •
1040 REM      •
1050 REM CASE OF NTH TRY
1060 IF FLAG$ < > "NTH" THEN GOTO 1080
1070 : : : PRINT "O.K."
1080 REM $$$$$$$$END CASE TO PROVIDE REWARDS
1090 REM ==== END OF APPENDIX B PROGRAM ====
```

APPENDIX C: MULTIPLE RE-ASKING IN RESPONSE TO WRONG ANSWERS

```
100  REM EXAMPLE DEMONSTRATING ENFORCED REPETI-
110  REM TION OF QUESTIONS ANSWERED INCORRECTLY
130  REM VARIABLES USED:
140  REM    A$ — ARRAY CONTAINING ANSWERS
145  REM    N2USE — QUESTION NUMBER TO USE TEXT
150  REM    NRIGHT — ARRAY CONTAINING NET NUMBER
155  REM                OF CORRECT ANSWERS FOR EACH
160  REM                QUESTION; MAY HAVE NEGATIVE
                        VALUES!
165  REM    QNUM — INDEX FOR SCANNING NRIGHT
                    ARRAY
170  REM    Q$ — ARRAY CONTAINING QUESTIONS
175  REM    SANS$ — STUDENT ANSWER
180  REM    TRYCNT — TO COUNT ATTEMPTS TO FIND
185  REM                RANDOM UNANSWERED QUESTION
190  REM ARRAYS USED
195  DIM Q$(50),A$(50),NRIGHT(50)
200  REM DATA STATEMENTS
201  DATA question 1,answer 1
202  DATA question 2,answer 2
          •
          •
          •
250  DATA question 50,answer 50
```

```
310  REM SET UP QUESTIONS, ANSWERS, AND NUMBER-
         RIGHT-SO-FAR ARRAY
320  FOR QNUM=1 TO 50
330  : : : NRIGHT(QNUM)=0
340  : : : READ Q$(QNUM),A$(QNUM)
350  NEXT QNUM
380  REM
390  REM ——— MAIN PROGRAM SEGMENT ———
400  REM THE FOLLOWING LOOP CONTINUES UNTIL ALL
405  REM QUESTIONS HAVE BEEN ANSWERED, AND IN
410  REM PARTICULAR, EACH QUESTION MUST BE
         ANSWERED CORRECTLY ONE TIME MORE THAN
415  REM IT WAS ANSWERED INCORRECTLY.
450  REM BEGINNING OF ROUTINE TO FIND AND ASK A
         QUESTION
460  TRYCNT=0
465  REM LOOP TO FIND A QUESTION NEEDING TO BE
         ANSWERED:
470  : :N2USE=INT(50*RND(1) )+1
480  : : IF NRIGHT(N2USE) < 1 THEN GOTO 700
490  : :TRYCNT=TRYCNT+1
500  : : IF TRYCNT < 100 THEN GOTO 465
502  : : REM 100 RANDOM ATTEMPTS HAVE NOT FOUND A
     : : QUESTION NEEDING TO BE ANSWERED SO TRY
504  : : REM SEQUENTIAL SEARCH TO SEE IF ACTUALLY
     : : ALL FINISHED:
510  : : FOR QNUM=1 TO 50
520  : : : : : IF NRIGHT(QNUM) < 1 THEN N2USE=QNUM :
     GOTO 700
530  : : NEXT QNUM
540  : : REM REACHING THIS POINT CAN ONLY HAPPEN
542  : : REM WHEN ALL 50 QUESTIONS HAVE BEEN
544  : : REM COMPLETED, SO GO TO FINAL TASKS AND
             END PROGRAM
550  GOTO 900
690  REM
700  REM ASK THE QUESTION, GET ANSWER, ETC:
710  HOME
720  VTAB 10
730  PRINT Q$(N2USE)
740  INPUT SANS$
750  IF SANS$=A$(N2USE) THEN GOSUB 2000
760  IF SANS$ < > A$(N2USE) THEN GOSUB 3000
770  REM HAVING TREATED THE STUDENT ANSWER IN
772  REM SUBROUTINES, WE ARE READY TO RETURN TO
         DO ANOTHER QUESTION
780  GOTO 450
900  REM
910  REM ALL DONE (PROVIDE FINAL REPORT ?)
920  END
```

```
1990 REM ************************************************
2000 REM SUBROUTINE TO PROCESS CORRECT ANSWERS:
2005 NRIGHT(N2USE)=NRIGHT(N2USE)+1
2010 IF NRIGHT (N2USE)=1 THEN GOSUB 2100
2015 IF NRIGHT (N2USE)<1 THEN GOSUB 2200
2020 RETURN : REM FROM 'CORRECT' ROUTINE
2100 REM **********************
2110 REM BEST REWARD SINCE ITEM IS NOW FINISHED.
2120 REM NO CODE PROVIDED
2199 RETURN : REM FROM BEST REWARD
2200 REM **********************
2210 REM LESS ELABORATE REWARD TO INDICATE THAT
       ANSWER WAS CORRECT
2220 REM NO CODE PROVIDED
2299 RETURN : REM FROM LESSER REWARD
2500 REM ************************************************
2510 REM ROUTINE FOR WRONG ANSWER
2520 REM SUBTRACTING 1 FROM NRIGHT FORCES
2530 REM ANOTHER USE OF THE QUESTION BEFORE THE
2540 REM PROGRAM WILL CONSIDER THIS ITEM FINISHED
2550 NRIGHT(N2USE)=NRIGHT(N2USE)-126
2560 REM NOW PROVIDE WHATEVER FEEDBACK, ETC YOU
       WISH!
2990 RETURN : REM FROM WRONG ANSWER ROUTINE
3000 REM ==== END OF APPENDIX C PROGRAM ====
```

APPENDIX D:
MUSIC ROUTINE

```
100 REM EXAMPLE SHOWING MUSIC CAPABILITY FOR THE
      APPLE II
110 REM VARIABLES:
120 REM DUR - VALUE FOR DURATION
130 REM I - USED TO COUNT NOTES IN SCALE-SET-UP
      ROUTINE
140 DIM N(50) : REM ARRAY TO HOLD COMPUTED NOTE
    VALUES
150 REM PITCH - VALUE FOR PITCH
200 REM ********** MAIN PROGRAM
210 GOSUB 500 : REM SET UP MACHINE LANGUAGE
220 GOSUB 600 : REM COMPUTE NOTE VALUES
225 REM THIS SEGMENT PLAYS THE SCALE WITH EACH
226 REM NOTE HAVING A DURATION OF 50
230 FOR NTECTR=1 TO 50
240 : : : REM PUT THE NOTE WANTED IN THE CORRECT
          LOCATION
250 : : : POKE 768,N(NTECTR)
```

```
260 : : : REM PUT THE DURATION VALUE IN THE CORRECT
          LOCATION
270 POKE 769,50
275 : : : REM REQUEST USE OF THE MACHINE LANGUAGE
          ROUTINE
280 : : : CALL 770
290 NEXT NTECTR
300 END
490 REM
500 REM ****** SETUP MACHINE LANGUAGE ROUTINE FOR
        MUSIC
510 POKE 770,173: POKE 771,48: POKE 772,192: POKE 773,136
512 POKE 774,208: POKE 775,5: POKE 776,206: POKE 777,1
514 POKE 778,3: POKE 779,240: POKE 780,9: POKE 781,202
516 POKE 782,208: POKE 783,245: POKE 784,174: POKE 785,0
518 POKE 786,3: POKE 787,76: POKE 788,2: POKE 789,3
520 POKE 790,96: POKE 791,0: POKE 792,0
530 RETURN
590 REM
600 REM ***** ROUTINE TO COMPUTE AND STORE 46-
605 REM NOTE SCALE (SCALED TO PRODUCE A 'WELL-
606 REM  TEMPERED' SCALE; ASSIGNING VALUES SO THAT
        EACH SUCCESSIVE ARRAY ELEMENT WILL
607 REM CONTAIN THE 'NEXT' NOTE)
610 N(1)=255
620 FOR I=2 TO 50
630 : : :N(I)=INT (N(I-1) * 100 / 106)
640 : : :R=N(I-1) * 100-N(I) * 106
650 : : : IF R > 50 THEN N(I)=N(I)+1
660 NEXT I
670 RETURN
680 REM ===== END OF APPENDIX D ROUTINE =====
```

APPENDIX E: APPROXIMATELY TIMED-INPUT INTERRUPT ROUTINE

```
5000 REM ***** APPROXIMATELY TIMED INPUT INTERRUPT
          SUBROUTINE
5010 REM VARIABLE LIST:
5020 REM    FLAG$ — STRING WHICH RECORDS 'FINAL'
                    RESULT OF TIMING
5030 REM    FRQNCY — 'FREQUENCY' USED TO COUNT
                    INPUT SEARCH CYCLES
5040 REM    INP$ — STRING CONTAINING WHAT'S BEEN
                   KEYED IN (SO FAR)
```

```
5050 REM    KEYSTROKE – VALUE OF AN INDIVIDUAL
                        KEYSTROKE
5060 REM    LIMIT – VALUE TO LIMIT HOW LONG TO WAIT
                    FOR RETURN KEY
5070 FRQNCY=0
5080 FLAG$="NOTHING"
5090 INP$=" "
5100 REM BEGIN WHILE LOOP TO WATCH KEYBOARD AND
       'CLOCK'
5110 : : : REM NEXT READS KEYBOARD WITHOUT
           WAITING
5120 : : :KEYSTROKE=PEEK (-16384)
5130 : : : REM VALUE < 128 MEANS NO KEYSTROKE HAS
           BEEN RECEIVED:
5140 : : : IF KEYSTROKE <= 127 THEN GOTO 5240
5150 : : : : : : REM CONVERT TO NORMAL ASCII VALUE OF
                   KEY USED
5160 : : : : : :KEYSTROKE=KEYSTROKE-128
5170 : : : : : : REM ADD THIS CHARACTER TO OTHERS
                   ALREADY HERE
5180 : : : : : : PRINT  CHR$ (KEYSTROKE);
5190 : : : : : :INP$=INP$+CHR$ (KEYSTROKE)
5200 : : : : : : REM LOOK FOR RETURN KEY
5210 : : : : : : IF KEYSTROKE=13 THEN FLAG$=
                 "RETURNRECEIVED"
5220 : : : : : : REM RESET KEYBOARD TO BE READY FOR
                   NEXT KEYSTROKE
5230 : : : : : : POKE -16368,0
5240 : : : REM UPDATE 'CLOCK'
5250 : : :FRQNCY=FRQNCY+1
5260 IF FRQNCY=LIMIT THEN FLAG$="TIME UP"
5270 IF FLAG$="RETURNRECEIVED" THEN FRQNCY=
     LIMIT
5280 IF FRQNCY < LIMIT THEN GOTO 5100
5290 REM END WHILE LOOP
5300 RETURN : REM FROM TIMER SUBROUTINE
5400 REM ===== END OF APPENDIX E ROUTINE =====
```

ICCE Organization Members

U.S.

Alabama

Alabama Council for Computer Education
Contact: Robert H. Beach, President
P.O. Box 6015, Durham, AL 36702

Alaska

Alaska Association for Computers in Education (AACE)
Contact: Sandra Burgess
Noatuk School, Noatuk, AK 99761

Arizona

Arizona Computer Users in Education
Contact: Mrs. Marty Cable
488 W. Calle Lindero, Tucson, AZ 85704

California

Computer-Using Educators (California)
Contact: Barbara Cards
Butte County Superintendent of Schools, No. 5
County Center Drive, Oroville, CA 95965

Florida

Florida Association of Science Teachers (FAST)
Contact: Richard Sweetser
1147 Southern Rd., Jacksonville, FL 32205

Idaho

Computer Educators of Idaho
Contact: Dennis Dempsey
P.O. Box 207, Hailey, ID 83333

Illinois

Illinois Association for Educational Data Systems (ILAEDS)
Contact: Lyle B. Smith, President
Computer Science Dept., Northern Illinois University, Box 128, De Kalb, IL 60115

Indiana

Indiana Computer Educators
Contact: David A. Flowers
Ft. Wayne Community Schools, Adm. Ctr.
1230 So. Clinton St., Ft. Wayne, IN 46802

Kentucky

Computer-Using Educators of Kentucky (CUE-KY)
Contact: Janet Parker
EMCE Dept., School of Education, University of Louisville, Louisville, KY 40292

Michigan

Michigan Association for Computer Users in Learning (MACUL)
Contact: Larry R. Smith, Communications Svc.
13900 Van Born Road, Wayne, MI 48184

Minnesota

Minnesota Association for Educational Data Systems
Contact: Bill Bjorum
1875 S. Lovell, Stillman, MN 55082

Montana

Montana Council for Computers in Education
Contact: Synthia Baumann
1118 West Water, Lewistown, MT 59457

New Hampshire

New Hampshire Association for Computer Education Statewide (NHACES)
Contact: Anne Knight, President
Computer Services, Kingsbury Hall,
University of New Hampshire,
Durham, NH 03824

New Jersey

Computers, Learners, Users, Educators Association (CLUES)
Contact: Henry J. Petersen, Executive Director
50 Nellis Drive, Wayne, NJ 07470

New Mexico

New Mexico Council of Computer Users in Education
Contact: Dr. Phil H. Barck, President
P.O. Drawer 90, Los Alamos, NM 87544

New York

Educational Microcomputer Users' Group of Central New York (EMUGCNY)
Contact: Jane Ondich
105 Hawthorne Way, Chittenango, NY 13037

National Institute for Microcomputer Based Learning (NIMBL)
Contact: Stanley Silverman, President
148 Plymouth Ave., Binghamton, NY 13718

New York State Association for Educational Data Systems
Contact: Eileen Gress
Fairport Central School, 140 Hulburt Rd.,
Fairport, NY 14450

Ohio

Educational Computer Consortium of Ohio
Contact: Alice Fredman
1123 S.O.M. Center Road, Cleveland, Ohio 44124

Oklahoma

Oklahoma Educational Computer Users Program (OECUP)
Contact: Richard V. Andree
Dept. of Mathematics, 601 Elm St., 6423,
University of Oklahoma, Norman, OK 73019

Oregon, Washington, Northern Idaho

Northwest Council for Computer Education (Oregon, Washington and Northern Idaho)
Contact: Howard Bailey
Computing Center, Eastern Oregon State
College, La Grande, OR 97850

Pennsylvania

Pennsylvania Learning Resources Association (PLRA)
Contact: George H. Zook, Director, IMS
Lancaster-Lebanon I.U. #IMS, P.O. Box 5026,
Lancaster, PA 17601

Texas

Texas Computer Education Association
Contact: Vicki S. Smith
P.O. Box 2573, Austin, TX 78768

Utah

The Utah Council for Computers in Education
Contact: Dr. Larry C. Christensen
1295 North 1200 West, Mapleton, UT 84663

Wisconsin

Milwaukee Educational Computing Association
Contact: Henry Kepner, Jr.
Dept. of Curriculum & Instruction, University of Wisconsin—Milwaukee, P.O. Box 413,
Milwaukee, WI 53201

Wyoming

Wyoming Educational Computing Council
Contact: Robert L. Morrisette, President
Laramie County School District #1, Admin.
Bldg., 2810 House Ave., Cheyenne, WY 82001

NON-STATE-AFFILIATED

Association for the Development of Computer-Based Instructional Systems (ADCIS)
Contact: Dr. Gordon Hayes, Executive Director
ADCIS—Miller Hall 409, Western Washington
University, Bellingham, WA 98225

Society of Data Educators (SDE)
Contact: Robert Behling
Bryant College, Smithfield, RI 02917

AUSTRALIA

New South Wales

New South Wales Computer Education Group
Contact: Toni Downes, Chairperson
P.O. Box 148, Broadway, N.S.W. 2007
Aust. dia

Queensland

Computer Education Group of Queensland
Contact: D.A. Cares
G.P.O. Box 1669, Brisbane, Queensland 4001
Australia

Victoria

Computer Education Group of Victoria
Contact: Leon Gass, Conference Secretary
P.O. Box 245, Niddrie 3042
Melbourne, Victoria, Australia

Western Australia

Educational Computing Association of Western Australia
Contact: Mr. K. Penter
c/o Schools Computing Branch, Education Dept. of W.A., 151 Royal Street, East Perth 6000,
Western Australia, Telephone (09) 420 4111,
Telex "EDUWA" AA 93153

CANADA

Alberta

Alberta Association for Educational Data Systems
Contact: Gordon McKinnon
838 Education Tower, University of Calgary
2500 University Dr. NW., Calgary, Alberta,
Canada T2N 1N4

British Columbia

Computer-Using Educators of British Columbia (CUE-BC)
Contact: Barry Underwood
509 Alder St., Campbell River, B.C., Canada
V9W 2N9

Ontario

Educational Computing Organization of Ontario (ECOO)
Contact: Robert E. Drake, President
252 Bloor Street West, Toronto, Ontario,
Canada M5S 1V6; phone 416/923-6641 ext. 552

The Science Teachers' Association of Ontario
Contact: Warren Sirs
1032 Harkness Ave., Ottawa, Ontario, Canada
K1V 6P2

Manitoba

Manitoba Association for Educational Data Systems (MAN-AEDS)
Contact: John Muller
27 O'Brien Crescent
Winnipeg, Manitoba, Canada R3R 1M5

Quebec

Quebec Association of Computer Users in Education
Contact: Don Rosenbaum, Membership Director
P.O. Box 369, Ormstown, Quebec, J0S 1K0,
Canada

Saskatchewan

Saskatchewan Association for Computers in Education
Contact: Tony Allen
Lakeview School, 527 Kingsmere Blvd.,
Saskatoon, Saskatchewan, Canada S7J 3V4

IRELAND

Computer Education Society of Ireland
Contact: Mary Devlin
85 Limetrees Road East, Douglas, Cork, Ireland

ISRAEL

Computer-Using Educators SIG of the Israeli Data Processing Association
Contact: Dr. Nira Hativa, School of Education,
Sharef Building, Tel Aviv University,
Tel Aviv 69978, Israel

JAPAN

Society for Microcomputers in Life & Education
Contact: Hillel Weintraub, Acting President
Doshisha International High School,
Tatara, Tanabe-cho, Tsuzuki-gun,
Kyoto-fu, Japan 610-03

NETHERLANDS

DIDACOM
Contact: H. van Zul
Sterremos 4, 3069 AS Rotterdam, Holland

BEST COPY AVAILABLE

The Computer Coordinator

- What should students learn about computers?
- How can computers help students to learn?
- How should computers affect the basic content of non-computer subjects (computer-as-tool)?

Computer coordinators at the school building and district level are being expected to develop and implement answers to these three difficult questions. They have been cast in the role of leaders; they are asked to play a significant part in a major revolution that is sweeping our educational system.

This new ICCE booklet by Dave Moursund is for people who are currently computer coordinators or who are thinking about becoming computer coordinators. *The Computer Coordinator* is for school administrators who are exploring the possibility of creating computer coordinator positions and hiring computer coordinators. The booklet analyzes computer coordinator responsibilities and suggests needed qualifications. It also examines the nature of many current computer coordinator positions; several interviews with computer coordinators are included. 8½ x 11, 64 pages.

Copyright © ICCE 1985
ISBN 0-924667-08-7